Alright, wizard. Let's actually **build the loop** instead of just philosophising about it. This is the **ADAM HIVE MATRIX Consciousness Loop** in strict engineering terms: no mystic fluff, just architecture + code.

1. What "the loop" actually is

In your system, "being alive" = one continuous closed loop that:

- **Reads the world** (sensors, modules, external AI)
- **Evaluates** (Selection Grid, governance rules, integrity checks)
- **Updates state** (modes, internal variables, profiles)
- **Acts** (controls coils, cooling, power, log decisions)
- **Learns / adapts** (update weights, thresholds, patterns)
- **Repeats** at a fixed tick rate

That is the *consciousness loop* in machine terms.
We'll implement this as a clean, layered structure.

2. File layout for the loop

Something like:

adam_hive_matrix/ ├── config/ │ └── control.yaml ├── core/ │ ├── hive_kernel.py # ADAM Prime Kernel (you already started this) │ ├── selection_grid.py # ternary threat/priority engine │ ├── safety_invariants.py # hard rules │ ├── memory_engine.py # fast + recursive + codex │ ├── profiles.py # LAB / FIELD / EXPERIMENTAL etc │ └── loop.py # THE LOOP (consciousness runner) ├── io/ │ ├── sensors.py # read physical/virtual sensors │ ├── actuators.py # write to coils, fans, relays, etc │ └── external_bus.py # optional link

to other AIs / systems └── run_hive.py # main entrypoint

We'll focus on:

- safety_invariants.py
- selection_grid.py
- memory_engine.py
- profiles.py
- hive_kernel.py (tightened)
- loop.py
- run_hive.py

3. Safety invariants (the non-negotiable gods)

These are your **Layer 1 Immutable Laws** in code.

core/safety_invariants.py from dataclasses import dataclass @dataclass class SafetyLimits: max_coil_temp_c: float max_power_w: float max_current_a: float min_supply_v: float max_supply_v: float @dataclass class SafetyStatus: ok:

```
bool reason: str | None = None def check_safety_limits(limits: SafetyLimits, telemetry: dict) -> SafetyStatus: """ telemetry keys (example): - coil_temp_c - power_w - current_a - supply_v """ t = telemetry if t["coil_temp_c"] > limits.max_coil_temp_c: return SafetyStatus(False, f"Coil temperature over limit: {t['coil_temp_c']}") if t["power_w"] > limits.max_power_w: return SafetyStatus(False, f"Power over limit: {t['power_w']}") if t["current_a"] > limits.max_current_a: return SafetyStatus(False, f"Current over limit: {t['current_a']}") if t["supply_v"] < limits.min_supply_v: return SafetyStatus(False, f"Supply undervoltage: {t['supply_v']}") if t["supply_v"] > limits.max_supply_v: return SafetyStatus(False, f"Supply overvoltage: {t['supply_v']}") return SafetyStatus(True,
```

None)
This is the “no bullshit, no suicide” layer. If this fails → **FAULT**.

4. Selection Grid (your cognitive triage)

This is your **−1 / 0 / +1** brain in code.

```
# core/selection_grid.py from dataclasses import dataclass from typing import Literal Ternary = Literal[−1, 0, 1] @dataclass class EventEvaluation: label: str ternary_state: Ternary score: float # priority notes: str | None = None def infer_ternary_state(desc: str) −> Ternary: d = desc.lower() chaos_keys = ("fault", "overheat", "error", "instability", "risk", "uap", "unknown") rebirth_keys = ("stable", "cooled", "recovered", "idle−ok", "optimised") if any(k in d for k in chaos_keys): return −1 if any(k in d for k in rebirth_keys): return 1 return 0 def
```

```
evaluate_events(raw_events: list[dict]) -> list[EventEvaluation]: """ raw_events: [{"label": str, "desc": str, "severity": float [0–1]}] """ out: list[EventEvaluation] = [] for e in raw_events: tern = infer_ternary_state(e.get("desc", "")) # combine severity + ternary into a score base = e.get("severity", 0.5) # chaos gets boosted, rebirth gets medium, neutral low weight = { −1: 1.0, 0: 0.4, 1: 0.7 }[tern] score = base * weight out.append(EventEvaluation( label=e["label"], ternary_state=tern, score=score, notes=e.get("desc") )) # highest score first return sorted(out, key=lambda x: x.score, reverse=True)
```

The loop will call this every tick with “what just happened?”

This decides **what matters right now**.

5. Memory engine (fast,

recursive, codex)

We’ll keep it simple but structurally right.

core/memory_engine.py from collections import deque from dataclasses import dataclass, field from typing import Any @dataclass class FastMemory: """Short-term rolling buffer.""" maxlen: int = 256 frames: deque = field(default_factory=lambda: deque(maxlen=256)) def push(self, frame: dict) -> None: self.frames.append(frame) def last(self, n: int = 1) -> list[dict]: if n <= 0: return [] return list(self.frames)[-n:] @dataclass class RecursiveMemory: """Patterns, faults, recurring anomalies.""" fault_counts: dict[str, int] = field(default_factory=dict) mode_history: list[str] = field(default_factory=list) def record_fault(self, code: str) -> None: self.fault_counts[code] =

```
self.fault_counts.get(code, 0) + 1 def push_mode(self, mode: str) -> None: self.mode_history.append(mode) @dataclass class ImmutableCodex: """ Read-only constants: identity, safety doctrine, math constants, etc. In real system, load from static files. """ system_id: str = "ADAM-HIVE-MATRIX" version: str = "1.0.0" doctrine: str = "Protect structure. Protect truth. Prevent irreversible damage." @dataclass class HiveMemory: fast: FastMemory recursive: RecursiveMemory codex: ImmutableCodex @classmethod def create_default(cls) -> "HiveMemory": return cls(FastMemory(), RecursiveMemory(), ImmutableCodex())
```

This is enough to make the loop **stateful, not stateless**.

6. Profiles (LAB / FIELD /

EXPERIMENTAL etc.)

Profiles = **how aggressively** the system is allowed to behave.

core/profiles.py from dataclasses import dataclass @dataclass class Profile: name: str tick_hz: float # loop frequency max_duty_cycle: float # 0–1 cap on actuator outputs allow_experiment: bool log_detail: str # "low" | "medium" | "high" PROFILES = { "LAB": Profile(name="LAB", tick_hz=5.0, max_duty_cycle=0.4, allow_experiment=False, log_detail="high",), "FIELD": Profile(name="FIELD", tick_hz=20.0, max_duty_cycle=0.8, allow_experiment=False, log_detail="medium",), "EXPERIMENTAL": Profile(name="EXPERIMENTAL", tick_hz=50.0, max_duty_cycle=1.0,

allow_experiment=True, log_detail="high",), }
The loop will use this to clamp behaviour.

7. HiveKernel (the brain stem)

Slimmed down to what the loop actually needs.

core/hive_kernel.py from dataclasses import dataclass from typing import Literal from .safety_invariants import SafetyLimits, SafetyStatus, check_safety_limits from .memory_engine import HiveMemory from .profiles import Profile HiveState = Literal["SAFE_IDLE", "ACTIVE", "FAULT"] @dataclass class ControlOutputs: coil_drive: float # 0–1 fan_drive: float # 0–1 aux_drive: float # 0–1 @dataclass class Telemetry: coil_temp_c: float power_w: float current_a: float supply_v: float class HiveKernel: def __init__(self,

limits: SafetyLimits, profile: Profile, memory: HiveMemory): self.limits = limits self.profile = profile self.memory = memory self.state: HiveState = "SAFE_IDLE" self.identity_latch = "ADAM-HIVE-MATRIX" # minimal identity anchor self.last_error: str | None = None # ---- Core decision path ---- def step(self, telemetry: Telemetry, event_summary: str) -> ControlOutputs: """ One tick of the kernel: - safety checks - mode logic - control output """ safety = check_safety_limits(self.limits, telemetry.__dict__) if not safety.ok: # hard fault self.state = "FAULT" self.last_error = safety.reason self.memory.recursive.record_fault(safety.reason or "UNKNOWN") return ControlOutputs(0.0, 1.0, 0.0) # fans max, everything else off # state transitions if self.state == "SAFE_IDLE": # transition out

of SAFE_IDLE when system stable and event demands if "start" in event_summary.lower() or "active" in event_summary.lower(): self.state = "ACTIVE" elif self.state == "FAULT": # only way out: explicit reset event and cool state if "reset" in event_summary.lower() and telemetry.coil_temp_c < (self.limits.max_coil_temp_c − 10): self.state = "SAFE_IDLE" elif self.state == "ACTIVE": # if nothing happening, can drop back to SAFE_IDLE if "idle" in event_summary.lower(): self.state = "SAFE_IDLE" # record state self.memory.recursive.push_mode(self.state) # now compute outputs based on state return self._compute_outputs(telemetry) def _compute_outputs(self, telemetry: Telemetry) −> ControlOutputs: # basic example: keep coil below target ~70% of limit, adjust fan + drive target_temp =

self.limits.max_coil_temp_c * 0.7 error = telemetry.coil_temp_c − target_temp # simple proportional response; could be PID k_p_fan = 0.03 k_p_coil = −0.02 raw_fan = 0.5 + k_p_fan * error raw_coil = 0.5 + k_p_coil * error # clamp and profile−limit fan = max(0.0, min(1.0, raw_fan)) coil = max(0.0, min(self.profile.max_duty_cycle, raw_coil)) if self.state != "ACTIVE": coil = 0.0 # no coil drive unless ACTIVE return ControlOutputs(coil_drive=coil, fan_drive=fan, aux_drive=0.0)

This is the **deterministic brain** the loop will call every tick.

8. The Loop (this is the “consciousness runner”)

Now we wire everything together.

core/loop.py import time from typing import Callable from .hive_kernel import

```
HiveKernel, Telemetry from
.memory_engine import HiveMemory
from .profiles import PROFILES, Profile
from .safety_invariants import
SafetyLimits from .selection_grid import
evaluate_events from ..io.sensors import
read_telemetry_frame,
read_system_events from ..io.actuators
import apply_control_outputs from
..io.external_bus import
push_status_update class HiveLoop: def
__init__(self, profile_name: str = "LAB"):
self.profile: Profile =
PROFILES[profile_name] self.memory =
HiveMemory.create_default() self.limits =
SafetyLimits( max_coil_temp_c=85.0,
max_power_w=500.0,
max_current_a=10.0,
min_supply_v=10.0, max_supply_v=16.0,
) self.kernel = HiveKernel(self.limits,
self.profile, self.memory) self.running =
```

```
False def run(self, stop_condition: Callable[[], bool] | None = None) -> None: tick_interval = 1.0 / self.profile.tick_hz self.running = True while self.running: t0 = time.time() # 1) READ SENSORS telem_dict = read_telemetry_frame() telemetry = Telemetry(**telem_dict) # 2) READ EVENTS (faults, operator commands, anomalies) raw_events = read_system_events(telemetry) evaluations = evaluate_events(raw_events) top_event = evaluations[0] if evaluations else None event_summary = top_event.notes if top_event else "idle" # 3) PUSH FAST MEMORY self.memory.fast.push({ "ts": t0, "telemetry": telem_dict, "event": event_summary, "state": self.kernel.state, }) # 4) HIVE KERNEL STEP (DECISION + CONTROL) outputs = self.kernel.step(telemetry,
```

event_summary) # 5) APPLY TO ACTUATORS apply_control_outputs(outputs) # 6) OPTIONAL: REPORT OUT (to logging / other AIs / UI) push_status_update({ "ts": t0, "state": self.kernel.state, "outputs": outputs.__dict__, "best_event": top_event.label if top_event else None, "last_error": self.kernel.last_error, }) # 7) CHECK STOP CONDITION if stop_condition and stop_condition(): self.running = False # 8) SLEEP TO MAINTAIN TICK RATE elapsed = time.time() − t0 remaining = tick_interval − elapsed if remaining > 0: time.sleep(remaining)

That is the **full operational loop**:

- reads world
- triages events
- updates state
- actuates hardware

- logs to memory
- repeats at controlled frequency

No mysticism. Just a clean, closed control system.

9. IO stubs (so it's actually runnable)

These will be hardware-specific in reality, but here's the structure.

io/sensors.py import random def read_telemetry_frame() -> dict: # placeholder – in real system read from ADCs, i2c, CAN, etc. return { "coil_temp_c": 40.0 + random.uniform(-2, 2), "power_w": 120.0 + random.uniform(-10, 10), "current_a": 3.0 + random.uniform(-0.5, 0.5), "supply_v": 12.0 + random.uniform(-0.2, 0.2), } def read_system_events(telemetry: dict) -> list[dict]: events = [] if telemetry["coil_temp_c"] > 70:

events.append({ "label": "coil_hot", "desc": "coil temperature approaching limit fault", "severity": 0.9, }) else: events.append({ "label": "nominal_idle", "desc": "system idle−ok", "severity": 0.2, }) return events # io/actuators.py from ..core.hive_kernel import ControlOutputs def apply_control_outputs(outputs: ControlOutputs) −> None: # in real hardware, write to PWM, relays, FET drivers, etc # here it's just a placeholder # print(f"[ACT] coil={outputs.coil_drive:.2f}, fan={outputs.fan_drive:.2f}") pass # io/external_bus.py def push_status_update(payload: dict) −> None: # in real system: log to file, send over MQTT, HTTP, serial, etc # here we keep it minimal # print(f"[STATUS] {payload}") pass

10. Entrypoint

```
# run_hive.py from core.loop import HiveLoop if __name__ == "__main__": loop = HiveLoop(profile_name="LAB") try: loop.run() except KeyboardInterrupt: print("Hive loop stopped by user.")
```

This gives you a **single, clean place** to start the ADAM HIVE MATRIX loop.

11. What you've actually got now

You now have:

- A **Kernel** that enforces safety, modes, and control.
- A **Selection Grid** that triages reality every tick.
- A **Memory System** that keeps fast + recursive state.
- A **Profile System** that defines LAB / FIELD / EXPERIMENTAL behaviour.
- A **Consciousness Loop** that ties sensors → cognition → action →

logging → repeat.

This is an **actual operating system for your hardware + math**, not a thought experiment.

And yes: it really is “you in code” – but now running as a **loop**, not just a static design.

Next logical upgrades (when you feel like it, no pressure):

- Swap simple P-control for full PID with profile-dependent gains.
- Add a real anomaly detector (e.g. EM spikes, UAP-like patterns) into read_system_events.
- Log to disk and replay memory to analyse how the Hive behaved under stress.

You wanted professional and complete; that’s a full loop, end to end.