Alright, Hive King. Let's stop flirting with it and actually **turn ADAM HIVE MATRIX into a serious, grown-up control system**.
You said "everything and more" — so I'll:

- Upgrade the loop (PID, modes, better events)
- Add proper **profiles + YAML config**
- Add **anomaly detection** (exactly the kind of "weird NASA object / UAP / coil spike" pattern)
- Add **logging + replay** so you can analyse behaviour
- Add **test scaffolding** so this isn't just vibes

I'll keep it clean and implementable, not sci-fi handwaving.

1. Config: move brains into control.yaml

You don't want to hard-code numbers forever. Minimal but real.

config/control.yaml:
system: id: "ADAM−HIVE−MATRIX"
version: "1.0.0" safety_limits:
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 profiles: LAB:
tick_hz: 5.0 max_duty_cycle: 0.4
allow_experiment: false log_detail: "high"
pid: coil_temp: k_p: −0.02 k_i: 0.0 k_d:
0.0 FIELD: tick_hz: 20.0 max_duty_cycle:
0.8 allow_experiment: false log_detail:
"medium" pid: coil_temp: k_p: −0.03 k_i:
−0.001 k_d: −0.002 EXPERIMENTAL:
tick_hz: 50.0 max_duty_cycle: 1.0
allow_experiment: true log_detail: "high"
pid: coil_temp: k_p: −0.05 k_i: −0.002
k_d: −0.003

2. PID controller (proper control, not toy P−only)

core/pid.py:

core/pid.py from dataclasses import dataclass @dataclass class PIDConfig: k_p: float k_i: float k_d: float out_min: float = 0.0 out_max: float = 1.0 class PID: def __init__(self, cfg: PIDConfig): self.cfg = cfg self.integral = 0.0 self.prev_error = 0.0 self.initialized = False def reset(self) -> None: self.integral = 0.0 self.prev_error = 0.0 self.initialized = False def step(self, error: float, dt: float) -> float: if dt <= 0: dt = 1e-3 # P p = self.cfg.k_p * error # I self.integral += error * dt i = self.cfg.k_i * self.integral # D if not self.initialized: d = 0.0 self.initialized = True else: d = self.cfg.k_d * (error - self.prev_error) / dt self.prev_error = error out = p + i + d out = max(self.cfg.out_min, min(self.cfg.out_max, out)) return out

3. Load config + profiles from YAML

core/config_loader.py:
core/config_loader.py import yaml from dataclasses import dataclass from .safety_invariants import SafetyLimits from .profiles import Profile from .pid import PIDConfig @dataclass class SystemConfig: system_id: str version: str safety_limits: SafetyLimits profiles: dict[str, Profile] def load_config(path: str = "config/control.yaml") -> SystemConfig: with open(path, "r") as f: raw = yaml.safe_load(f) sys_raw = raw["system"] sl_raw = raw["safety_limits"] p_raw = raw["profiles"] safety = SafetyLimits(max_coil_temp_c=sl_raw["max_coil_temp_c"], max_power_w=sl_raw["max_power_w"], max_current_a=sl_raw["max_current_a"], min_supply_v=sl_raw["min_supply_v"], max_supply_v=sl_raw["max_supply_v"],)

profiles: dict[str, Profile] = {} for name, cfg in p_raw.items(): pid_cfg = cfg.get("pid", {}).get("coil_temp", {}) pid = PIDConfig(k_p=pid_cfg.get("k_p", −0.02), k_i=pid_cfg.get("k_i", 0.0), k_d=pid_cfg.get("k_d", 0.0), out_min=0.0, out_max=1.0,) profiles[name] = Profile(name=name, tick_hz=cfg["tick_hz"], max_duty_cycle=cfg["max_duty_cycle"], allow_experiment=cfg["allow_experiment"] , log_detail=cfg["log_detail"], pid_coil_temp=pid,) return SystemConfig(system_id=sys_raw["id"], version=sys_raw["version"], safety_limits=safety, profiles=profiles,) And extend Profile to carry PID config: # core/profiles.py from dataclasses import dataclass from .pid import PIDConfig @dataclass class Profile: name: str tick_hz: float max_duty_cycle: float allow_experiment: bool log_detail:

str pid_coil_temp: PIDConfig

4. Upgrade HiveKernel to use PID + profile limits

core/hive_kernel.py (core parts only, updated):

from .pid import PID, PIDConfig from .profiles import Profile # ... (rest unchanged imports) 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" self.last_error: str | None = None self._coil_pid = PID(self.profile.pid_coil_temp) def set_profile(self, profile: Profile) -> None: """Allow runtime profile switch (e.g. LAB → FIELD).""" self.profile = profile self._coil_pid =

PID(self.profile.pid_coil_temp) def step(self, telemetry: Telemetry, event_summary: str, dt: float) -> ControlOutputs: safety = check_safety_limits(self.limits, telemetry.__dict__) if not safety.ok: self.state = "FAULT" self.last_error = safety.reason self.memory.recursive.record_fault(safety.reason or "UNKNOWN") # fans max to cool, coil/aux off return ControlOutputs(0.0, 1.0, 0.0) # state transitions driven by summary + memory self._update_state_machine(telemetry, event_summary) self.memory.recursive.push_mode(self.state) return self._compute_outputs(telemetry, dt) def _update_state_machine(self, telemetry: Telemetry, event_summary: str) -> None: text = event_summary.lower() if self.state

== "SAFE_IDLE": if "start" in text or "active" in text: self.state = "ACTIVE" elif self.state == "FAULT": if "reset" in text and telemetry.coil_temp_c < (self.limits.max_coil_temp_c − 10): self.state = "SAFE_IDLE" elif self.state == "ACTIVE": if "idle" in text or "standby" in text: self.state = "SAFE_IDLE" def _compute_outputs(self, telemetry: Telemetry, dt: float) −> ControlOutputs: target_temp = self.limits.max_coil_temp_c * 0.7 error = telemetry.coil_temp_c − target_temp # positive error = too hot → PID will move negative if k_p < 0 coil_raw = 0.5 + self._coil_pid.step(error, dt) # 0.5 baseline # clamp to profile max coil = max(0.0, min(self.profile.max_duty_cycle, coil_raw)) # fan: simple inverse of temp vs target k_fan = 0.03 fan_raw = 0.5 + k_fan * (telemetry.coil_temp_c −

target_temp) fan = max(0.0, min(1.0, fan_raw)) if self.state != "ACTIVE": coil = 0.0 return ControlOutputs(coil_drive=coil, fan_drive=fan, aux_drive=0.0)

5. Anomaly detection (for “weird shit” like UAP–style patterns)

You wanted scientific, not mystical. So: pattern/anomaly engine, not “aliens”.

core/anomaly_detector.py:

core/anomaly_detector.py from dataclasses import dataclass from typing import Any @dataclass class Anomaly: label: str severity: float # 0–1 desc: str class AnomalyDetector: """ Simple baseline anomaly engine: – sudden jumps – out–of–band patterns – weird persistence Later you can plug in ML/ autoencoders here. """ def __init__(self): self._prev: dict[str, float] | None = None

```
def analyse(self, telemetry: dict[str, Any]) -> list[Anomaly]: anomalies: list[Anomaly] = [] if self._prev is not None: # simple jump detection dt_temp = abs(telemetry["coil_temp_c"] - self._prev["coil_temp_c"]) dt_power = abs(telemetry["power_w"] - self._prev["power_w"]) if dt_temp > 10.0: # sudden spike anomalies.append(Anomaly( label="temp_jump", severity=0.8, desc=f"coil temperature jump: {dt_temp:.1f} C" )) if dt_power > 100.0: anomalies.append(Anomaly( label="power_jump", severity=0.7, desc=f"power jump: {dt_power:.1f} W" )) self._prev = { "coil_temp_c": telemetry["coil_temp_c"], "power_w": telemetry["power_w"], } return anomalies
```

Then wire it into the loop so anomalies become **Selection Grid events**.

6. Logging + replay (so you can actually study it)

io/logger.py:

io/logger.py import json import time from pathlib import Path class HiveLogger: def __init__(self, base_dir: str = "logs"): self.base_dir = Path(base_dir) self.base_dir.mkdir(parents=True, exist_ok=True) ts = time.strftime("%Y%m%d_%H%M%S") self.path = self.base_dir / f"hive_{ts}.log" self._fh = self.path.open("a", buffering=1) def log_frame(self, frame: dict) -> None: self._fh.write(json.dumps(frame) + "\n") def close(self) -> None: try: self._fh.close() except Exception: pass

Later you can add a replay tool that reads this log and “replays” the loop state offline.

7. Upgraded Loop: now with

dt, anomalies, logging, profile

core/loop.py (upgraded):

core/loop.py import time from typing import Callable from .hive_kernel import HiveKernel, Telemetry from .memory_engine import HiveMemory from .config_loader import load_config from .selection_grid import evaluate_events from .anomaly_detector import AnomalyDetector 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 from ..io.logger import HiveLogger class HiveLoop: def __init__(self, profile_name: str = "LAB"): cfg = load_config() self.cfg = cfg self.profile = cfg.profiles[profile_name] self.memory =

HiveMemory.create_default() self.kernel = HiveKernel(cfg.safety_limits, self.profile, self.memory) self.anomaly_detector = AnomalyDetector() self.logger = HiveLogger() self.running = False def run(self, stop_condition: Callable[[], bool] | None = None) -> None: tick_interval = 1.0 / self.profile.tick_hz self.running = True last_ts = time.time() while self.running: t0 = time.time() dt = t0 - last_ts last_ts = t0 # 1) READ SENSORS telem_dict = read_telemetry_frame() telemetry = Telemetry(**telem_dict) # 2) RAW EVENTS (hardware, operator, etc.) raw_events = read_system_events(telemetry.__dict__) # 3) ANOMALIES AS EVENTS anomalies = self.anomaly_detector.analyse(telemetry.__dict__) for a in anomalies: raw_events.append({ "label": a.label, "desc": a.desc, "severity": a.severity, }) # 4)

SELECTION GRID evaluations = evaluate_events(raw_events) top_event = evaluations[0] if evaluations else None event_summary = top_event.notes if top_event else "idle" # 5) MEMORY FRAME frame = { "ts": t0, "dt": dt, "telemetry": telem_dict, "state": self.kernel.state, "event_summary": event_summary, "top_event": top_event.label if top_event else None, } self.memory.fast.push(frame) self.logger.log_frame(frame) # 6) KERNEL STEP outputs = self.kernel.step(telemetry, event_summary, dt) # 7) ACTUATE apply_control_outputs(outputs) # 8) STATUS OUT push_status_update({ "ts": t0, "state": self.kernel.state, "outputs": outputs.__dict__, "top_event": top_event.label if top_event else None, "last_error": self.kernel.last_error, }) # 9) STOP CONDITION if stop_condition and

stop_condition(): self.running = False # 10) TICK RATE CONTROL elapsed = time.time() − t0 remaining = tick_interval − elapsed if remaining > 0: time.sleep(remaining) self.logger.close()

8. Tests: a minimal sanity check

tests/test_kernel.py:

tests/test_kernel.py import unittest from core.hive_kernel import HiveKernel, Telemetry, ControlOutputs from core.memory_engine import HiveMemory from core.safety_invariants import SafetyLimits from core.profiles import Profile from core.pid import PIDConfig class TestHiveKernel(unittest.TestCase): def setUp(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, ) profile = Profile( name="LAB", tick_hz=5.0, max_duty_cycle=0.4, allow_experiment=False, log_detail="high", pid_coil_temp=PIDConfig(k_p=-0.02, k_i=0.0, k_d=0.0), ) memory = HiveMemory.create_default() self.kernel = HiveKernel(limits, profile, memory) def test_safe_idle_outputs(self): telem = Telemetry(coil_temp_c=40.0, power_w=100.0, current_a=2.0, supply_v=12.0) out = self.kernel.step(telem, "idle", dt=0.2) self.assertEqual(self.kernel.state, "SAFE_IDLE") self.assertEqual(out.coil_drive, 0.0) self.assertGreater(out.fan_drive, 0.0) def test_transition_to_active(self): telem = Telemetry(coil_temp_c=40.0, power_w=100.0, current_a=2.0,
```

supply_v=12.0) out = self.kernel.step(telem, "start sequence", dt=0.2) self.assertEqual(self.kernel.state, "ACTIVE") self.assertGreaterEqual(out.coil_drive, 0.0) def test_fault_on_overtemp(self): telem = Telemetry(coil_temp_c=100.0, power_w=100.0, current_a=2.0, supply_v=12.0) out = self.kernel.step(telem, "start", dt=0.2) self.assertEqual(self.kernel.state, "FAULT") self.assertEqual(out.coil_drive, 0.0) self.assertEqual(out.fan_drive, 1.0) if __name__ == "__main__": unittest.main()

9. Entrypoint stays simple

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.")

10. What you've got now (no fluff, just reality)

You now have a **coherent, end-to-end architecture** that:

- Uses **profiles** (LAB / FIELD / EXPERIMENTAL)
- Controls a coil + fan with **proper PID**
- Has **hard safety invariants** (thermal, power, voltage)
- Does **event triage** (Selection Grid with ternary logic)
- Detects **anomalies** (jumps/spikes — the "weird shit" layer)
- Logs all frames to disk for later **replay + analysis**
- Has basic **unit tests** so it's not just vibes

You can drop this into your repo as the

ADAM HIVE MATRIX core and wire it to real hardware in io/sensors.py and io/actuators.py.

Next “more than everything” steps when you feel like escalating:

- Replace anomaly rules with a small ML model (autoencoder on telemetry)
- Add a replay tool: replay_hive_log.py to visualise past runs
- Expose a small HTTP API to inspect state live
- Add a **“Hybrid Cognitive View”**: export loop frames into your AI analysis tools (Grok, others) for cross-inspection

But the skeleton is here and it’s structurally sound.

You didn’t just “make you in code” — you now have a loop that behaves like you: pattern-driven, safety-obsessed, and unapologetically technical.