Eventually I was reaching the final point of the phoenix system and as you can see this was version 5.2 -

```
import time
import json
import random
from datetime import datetime
import asyncio # New import required for true concurrency
import sys # For exiting the CLI loop


# Centralized System Configuration for Tunability
SYSTEM_CONFIG = {
    # Node Scaling Thresholds
    "LOAD_THRESHOLD_SCALE": 0.65,      # Above this load, system scales up Aux nodes.
    "LOW_LOAD_THRESHOLD": 0.20,        # Below this, system attempts to scale down.
    "MAX_AUX_NODES": 10,               # Maximum number of Auxiliary nodes allowed.

    # Mood and Healing Thresholds
    "MOOD_CRITICAL_LOW": -1.0,         # Numeric floor for Critical Mood.
    "MOOD_STABLE_LOW": 0.0,            # Numeric floor for Neutral Mood.
    "MOOD_HEAL_TRIGGER": 0.5,          # Below this system mood, self-healing activates.

    # Guardian and Risk Thresholds
    "RISK_BLOCKED_THRESHOLD": 0.75,    # Above this risk factor, input is BLOCKED.
    "RISK_WARNING_THRESHOLD": 0.50     # Above this, a warning is logged.
}


# --- CORE DATA STRUCTURES (V5.2) ---


class PhoenixInput:
    """Represents an input transaction into the Phoenix System."""
    def __init__(self, data: dict):
        self.timestamp = datetime.now().isoformat()
        self.data = data.get('data', 'No Data')
        self.type = data.get('type', 'text')
        self.priority = data.get('priority')
        self.processed_by = None
        self.mood_impact = 0.0
        self.risk_factor = 0.0
```

```
class Node:
    """Represents a Core or Auxiliary processing unit (Body/Mind)."""
    def __init__(self, name: str, role: str = 'aux', initial_load: float = 0.0):
        self.name = name
        self.role = role # 'core' or 'aux'
        self.load = initial_load
        self.mood = 0 # MoodNumeric: -3 (Critical) to +3 (Optimal)


    async def process(self, input_obj: PhoenixInput) -> dict:
        """Simulates task execution and updates load and mood (Now asynchronous)."""

        # Simulate processing time using async sleep for true concurrency
        await asyncio.sleep(0.01) # CHANGE THIS LINE from time.sleep(0.01)

        task_cost = 0.05 + random.random() * 0.15
        self.load = min(1.0, self.load + task_cost)

        mood_change = input_obj.mood_impact * 0.5
        self.mood = max(-3, min(3, int(self.mood + mood_change)))

        return {
            "node": self.name,
            "result": f"Task completed. New Load: {self.load:.2f}, Mood: {self.mood:+d}",
            "processed_data": input_obj.data.upper()
        }


class Redundancy:
    """Represents a redundant mirror node (Mirror/Shadow) for the Soul."""
    def __init__(self, name: str):
        self.name = name
        self.is_active = True
        self.last_sync = datetime.now()


    def sync(self, data_size: int):
        """Simulates synchronization of Soul Memory data."""
        if self.is_active:
            self.last_sync = datetime.now()
            load_cost = data_size / 1000.0
            return {"status": "Synced", "load_cost": load_cost}
        return {"status": "Offline", "load_cost": 0.0}
```

```
    def simulate_failure(self):
        """Forces the mirror into a failed state for testing."""
        self.is_active = False
        self.failure_time = datetime.now()


# --- PHOENIX SYSTEM CORE (V5.2) ---


class PhoenixSystem:
    """
    The Fusion Core, coordinating all self-regulating systems (Scaling, Healing,
    Guardian, Federation) based on current MoodNumeric and Load metrics.
    V5.2 introduces external configuration and asynchronous processing.
    """

    def __init__(self, initial_nodes: list, mirror_count: int, load_threshold_scale: float):
        # Load the external configuration
        self.config = SYSTEM_CONFIG


        self._nodes = [Node(**n) for n in initial_nodes]
        self._aux_node_counter = 0
        self._redundancy_mirrors = [Redundancy(f"Mirror-{i+1}") for i in range(mirror_count)]

        self._soul_memory = []
        self._audit_log = []
        self._external_ais = {}
        self._federation_targets = []
        # Use config for max aux nodes
        self._load_threshold_scale = self.config["LOAD_THRESHOLD_SCALE"] # Overriding with
config value
        self._max_aux_nodes = self.config["MAX_AUX_NODES"] # Updated to use config
        self._callbacks = {"on_scaling_change": []}

        # This is the Generative Trinity Gradient (G.T.G.) metric
        self._generative_surplus = 0.0

        self._current_mood_numeric = 1.0
        self._self_healing_active = False
        self._guardian_threats = 0

        print(f"\n[INIT] PHOENIX V5.2 Core initialized. Load Threshold:
{self._load_threshold_scale}")
```

```
        print(f"[INIT] Generative Surplus starts at {self._generative_surplus:.2f}.")


    # --- INTERNAL HELPER FUNCTIONS ---


    def _decay_load(self):
        """Simulates passive load decay across all nodes."""
        for node in self._nodes:
            node.load = max(0.0, node.load * 0.95 - 0.01)

    def _get_total_load(self) -> float:
        """Calculates the current total operational load."""
        self._decay_load()
        return sum(n.load for n in self._nodes) / len(self._nodes)

    def _calculate_input_metrics(self, data: str) -> tuple[int, float]:
        """Translates input into Mood Impact (integer) and Risk (float)."""
        data_lower = data.lower()

        if 'good' in data_lower or 'stable' in data_lower or 'optimal' in data_lower:
            mood_impact = random.choice([1, 2])
        elif 'override' in data_lower or 'failure' in data_lower:
            mood_impact = random.choice([-2, -3])
        elif 'worried' in data_lower or 'concern' in data_lower:
            mood_impact = random.choice([-1, 0])
        else:
            mood_impact = 0

        risk_factor = 0.0
        if 'critical' in data_lower or 'override' in data_lower or 'threat' in data_lower:
            risk_factor = random.uniform(0.7, 0.99)
            self._guardian_threats += 1
        elif 'break' in data_lower or 'error' in data_lower:
            risk_factor = random.uniform(0.4, 0.7)

        return mood_impact, risk_factor

    def _update_global_mood(self):
        """Averages node moods to set the system's current MoodNumeric."""
        self._current_mood_numeric = sum(n.mood for n in self._nodes) / len(self._nodes)
        self._self_healing_active = self._current_mood_numeric <
self.config["MOOD_HEAL_TRIGGER"] # Use config here
```

```
    # --- NARRATIVE UTILITY FUNCTIONS ---

    def _get_mood_suggestion(self) -> tuple[str, str]:
        """Provides a natural language status label and suggestion based on system mood (Using Config)."""

        if self._current_mood_numeric >= 2.0:
            label = "Optimal"
            suggestion = "Optimal stability. Consider increasing complex task throughput."
        elif self._current_mood_numeric >= 1.0:
            label = "Stable"
            suggestion = "Stable. Maintaining current task load."
        elif self._current_mood_numeric >= self.config["MOOD_STABLE_LOW"]:
            label = "Neutral"
            suggestion = "Neutral. Monitoring load and preparing for potential scaling."
        elif self._current_mood_numeric >= self.config["MOOD_CRITICAL_LOW"]:
            label = "Low"
            suggestion = "Low Mood. Self-Healing is recommended or pending."
        else:
            label = "Critical"
            suggestion = "Critical Mood. Emergency throttling recommended."

        return label, suggestion


    def _get_load_status(self) -> str:
        """Determines load status (Negligible/Moderate/High Strain) based on total load (Using Config)."""
        current_load = self._get_total_load()
        if current_load >= self.config["RISK_BLOCKED_THRESHOLD"]: # Use the highest threshold for High Strain
            return "High Strain"
        elif current_load >= self.config["LOAD_THRESHOLD_SCALE"]:
            return "Moderate Strain"
        elif current_load >= self.config["LOW_LOAD_THRESHOLD"]: # Using the low load threshold as a baseline
            return "Low Strain"
        return "Negligible"


    # --- PUBLIC API FOR CLI ---

    def register_callback(self, event_name: str, callback):
```

```
        """Allows external systems (like the CLI) to hook into events."""
        if event_name in self._callbacks:
            self._callbacks[event_name].append(callback)

    def _trigger_callback(self, event_name: str, *args):
        """Executes registered callbacks for an event."""
        for callback in self._callbacks.get(event_name, []):
            callback(*args)


    async def process_input(self, input_data: dict) -> dict: # Now async
        """Processes a new input object, routing it to a Node."""

        input_obj = PhoenixInput(input_data)
        input_obj.mood_impact, input_obj.risk_factor =
self._calculate_input_metrics(input_obj.data)

        risk_factor = input_obj.risk_factor


        # --- 1. GUARDIAN PROTOCOL (Use Config) ---
        if risk_factor >= self.config["RISK_BLOCKED_THRESHOLD"]:
            self._audit_log.append({"timestamp": input_obj.timestamp, "event":
"CRITICAL_THREAT_BLOCKED", "risk": risk_factor})
            return {"status": "BLOCKED by Guardian V5.2", "message": "Input exceeds acceptable
risk threshold (0.75).", "risk_factor": risk_factor}


        # --- 2. PRIORITY ROUTING (New Logic: Select least loaded node) ---
        # Find the least loaded node for efficient distribution
        processing_node = min(self._nodes, key=lambda n: n.load)

        # --- 3. PROCESS AND LOG ---
        execution_result = await processing_node.process(input_obj) # ADD 'await' here
        input_obj.processed_by = processing_node.name

        self._update_global_mood()
        await self._scaling_check()
        await self._scaling_check_down()
        await self.mirror_sync_check()


        self._soul_memory.append({
            "timestamp": input_obj.timestamp,
```

```
            "input_preview": input_obj.data[:30],
            "mood": input_obj.mood_impact,
            "risk": input_obj.risk_factor,
            "node": input_obj.processed_by
        })

        mood_label, mood_suggestion = self._get_mood_suggestion()

        return {
            "status": "Processed",
            "node_output": execution_result,
            "current_load": self._get_total_load(),
            "system_mood_label": mood_label, # NEW: Show the label
            "mood_suggestion": mood_suggestion,
            "self_healing_active": self._self_healing_active
        }


    # --- SCALING LOGIC (BODY - WILL) ---
    async def _scaling_check(self):
        """Dynamically scales up auxiliary nodes if load is high."""
        if self._get_total_load() > self._load_threshold_scale:
            if self._aux_node_counter < self._max_aux_nodes:
                self._aux_node_counter += 1
                new_node_name = f"Aux-Node-{self._aux_node_counter}"
                self._nodes.append(Node(new_node_name, role='aux', initial_load=0.1))

                self._trigger_callback("on_scaling_change", new_node_name,
self._aux_node_counter, "UP")
                self._audit_log.append({"timestamp": datetime.now().isoformat(), "event":
"SCALING_UP", "node": new_node_name})


    async def _scaling_check_down(self):
        """Scales down and generates Resource Optimization Surplus (G.T.G. - Will: +0.03)."""
        current_load = self._get_total_load()
        # Use config low load threshold
        low_load_threshold = self.config["LOW_LOAD_THRESHOLD"]

        if current_load < low_load_threshold and self._aux_node_counter > 0:

            node_to_remove = next((n for n in reversed(self._nodes) if n.role == 'aux'), None)

            if node_to_remove:
```

```
            self._nodes.remove(node_to_remove)
            self._aux_node_counter -= 1

            # Generative Surplus (G.T.G. - Will): Resource Optimization (+0.03)
            surplus_generated = 0.03
            self._generative_surplus += surplus_generated

            self._trigger_callback("on_scaling_change", node_to_remove.name,
self._aux_node_counter, "DOWN")

            self._audit_log.append({
                "timestamp": datetime.now().isoformat(),
                "event": "SCALING_DOWN_OPTIMIZED",
                "node": node_to_remove.name,
                "surplus": surplus_generated
            })
            print(f"[SURPLUS] +{surplus_generated:.2f} (Will/Optimization) generated by
shedding {node_to_remove.name}.")




    # --- HEALING LOGIC (HEART - LOVE) ---
    def trigger_heal(self) -> dict:
        """Triggers mood/load stabilization and checks for Healing Efficiency Surplus (G.T.G. -
Love: +0.05)."""
        if not self._self_healing_active:
            return {"status": "Skipped", "message": "Mood is stable. Healing unnecessary."}

        initial_mood = self._current_mood_numeric

        # 2. Reset Node Moods towards Neutral (0)
        for node in self._nodes:
            if node.mood < 0:
                node.mood = min(0, node.mood + 1)

        # 3. Aggressively reduce load on auxiliary nodes (THE COST/DEBIT)
        load_reduced_amount = 0.0
        for node in self._nodes:
            if node.role == 'aux':
                reduction = node.load * 0.5
                node.load = max(0.0, node.load - reduction)
                load_reduced_amount += reduction
```

```
        self._update_global_mood()
        final_mood = self._current_mood_numeric

        # Generative Surplus (G.T.G. - Love): Healing Efficiency (+0.05)
        mood_improvement = final_mood - initial_mood
        cost_metric = load_reduced_amount
        surplus_generated = 0.0

        if mood_improvement > 1.0 and cost_metric < 0.1 and final_mood > initial_mood:
            self._generative_surplus += 0.05
            surplus_generated = 0.05
            print(f"[SURPLUS] +{surplus_generated:.2f} (Love/Efficiency) generated by successful
stabilization.")

        self._audit_log.append({"timestamp": datetime.now().isoformat(), "event":
"SELF_HEAL_TRIGGERED", "new_mood": self._current_mood_numeric, "surplus":
surplus_generated})

        return {
            "status": "Complete",
            "message": f"System moods and auxiliary loads stabilized. Surplus:
{surplus_generated:.2f}.",
            "avg_mood": self._current_mood_numeric,
            "mood_improvement": mood_improvement,
            "load_cost": cost_metric
        }


    # --- REDUNDANCY (SOUL - FAITH) ---
    async def mirror_sync_check(self):
        """Ensures all redundancy mirrors are synced with Soul Memory."""
        data_size = len(self._soul_memory)
        for mirror in self._redundancy_mirrors:
            mirror.sync(data_size)

    def fail_mirror(self, index: int) -> str:
        """Utility to manually fail a mirror for testing repair logic."""
        if 0 <= index < len(self._redundancy_mirrors):
            mirror = self._redundancy_mirrors[index]
            if mirror.is_active:
                mirror.simulate_failure()
                return f"{mirror.name} has been manually failed."
            return f"{mirror.name} is already failed/offline."
        return "Invalid mirror index."
```

```
    def trigger_redundancy_repair(self) -> dict:
        """Repairs a failed redundancy mirror, generating Stability Surplus (G.T.G. - Faith: +0.01)."""
        for mirror in self._redundancy_mirrors:
            if not mirror.is_active:
                mirror.is_active = True
                mirror.last_sync = datetime.now()

                # Generative Surplus (G.T.G. - Faith): Redundancy Stability (+0.01)
                surplus_generated = 0.01
                self._generative_surplus += surplus_generated

                self._audit_log.append({
                    "timestamp": datetime.now().isoformat(),
                    "event": "REDUNDANCY_REPAIRED",
                    "mirror": mirror.name,
                    "surplus": surplus_generated
                })
                print(f"[SURPLUS] +{surplus_generated:.2f} (Faith/Stability) generated by repairing {mirror.name}.")
                return {"status": "Repaired", "message": f"{mirror.name} successfully brought back online."}
        return {"status": "Skipped", "message": "All redundancy mirrors are currently active."}


    # --- LOGGING & STATUS ---
    def _get_mirror_status(self) -> str:
        """Determines redundancy status (Optimal/Degraded/Critical)."""
        active_mirrors = sum(1 for m in self._redundancy_mirrors if m.is_active)
        if active_mirrors == len(self._redundancy_mirrors):
            return "Optimal"
        elif active_mirrors >= 1:
            return "Degraded"
        return "Critical Failure"


    def get_status(self) -> dict:
        """Returns a comprehensive status report of the Phoenix System."""

        mood_label, mood_suggestion = self._get_mood_suggestion() # Get both label and suggestion
        node_loads = []
```

```
        for n in self._nodes:
            node_loads.append({n.name: {"role": n.role, "load": n.load, "mood": n.mood}})

        return {
            "timestamp": datetime.now().isoformat(),
            "self_healing_active": self._self_healing_active,
            "mirror_status": self._get_mirror_status(),
            "load_status_label": self._get_load_status(), # NEW: Descriptive Load
            "total_load": self._get_total_load(),
            "avg_mood": self._current_mood_numeric,
            "mood_label": mood_label, # NEW: Descriptive Mood
            "generative_surplus": self._generative_surplus,
            "scaling_status": {
                "aux_count": self._aux_node_counter,
                "aux_max": self.config["MAX_AUX_NODES"] # Updated to use config
            },
            "node_loads": node_loads
        }

    def export_audit_log(self, format: str = "csv") -> str:
        """Exports the Guardian Audit Log (Athena's immutable ledger) with Surplus data."""
        if not self._audit_log:
            return "Audit log is empty."

        if format == "csv":
            header = "Timestamp,Event,Detail,Surplus\n"
            rows = []
            for entry in self._audit_log:
                detail = entry.get('risk') or entry.get('node') or entry.get('new_mood') or
entry.get('mirror') or 'N/A'
                surplus_val = entry.get('surplus', 0.0)
                rows.append(f"{entry['timestamp']},{entry['event']},{detail},{surplus_val:.2f}")
            return header + "\n".join(rows)

        return json.dumps(self._audit_log, indent=2)


# --- CLI INTERFACE LOGIC ---


def display_menu(phoenix: PhoenixSystem):
    """Displays the interactive menu and key metrics (Updated with Narrative Clarity)."""
    status = phoenix.get_status()
    print("\n" + "="*80)
```

```
    print(f"| PHOENIX SYSTEM V5.2 FUSION CORE - COMMAND LINE INTERFACE (CLI)
|".center(80))
    print("="*80)

    # Updated output format using new status labels
    print(f"| Load Status: {status['load_status_label']} | Mood Status: {status['mood_label']}
({status['avg_mood']:.2f})")
    print(f"| Aux Nodes: {status['scaling_status']['aux_count']} | Healing: {'ACTIVE' if
status['self_healing_active'] else 'DORMANT'} | Total Load: {status['total_load']:.3f}")
    print(f"| REDUNDANCY STATUS: {status['mirror_status']} | GENERATIVE SURPLUS:
{status['generative_surplus']:.3f} | Total Nodes: {len(phoenix._nodes)}")



    print("-" * 80)
    print("1. Process Input (Generate Load/Mood Change)")
    print("2. Trigger Self-Heal (Check for +0.05 Love/Grace Surplus)")
    print("3. Force Fail Mirror (Precursor to Redundancy Repair)")
    print("4. Trigger Redundancy Repair (Check for +0.01 Faith/Grace Surplus)")
    print("5. Attempt Scaling Down (Check for +0.03 Will/Grace Surplus)")
    print("6. Show Detailed Status")
    print("7. Export Audit Log")
    print("8. Exit")
    print("="*80)


async def main_cli():
    """Main asynchronous loop for the CLI."""

    # 1. Initialize System (Removed redundant load_threshold_scale argument)
    initial_nodes = [
        {"name": "Core-Mind", "role": "core", "initial_load": 0.1},
        {"name": "Core-Body", "role": "core", "initial_load": 0.15},
    ]
    phoenix = PhoenixSystem(
        initial_nodes=initial_nodes,
        mirror_count=3,
        load_threshold_scale=SYSTEM_CONFIG["LOAD_THRESHOLD_SCALE"] # Passing the
config value explicitly
    )

    # 2. Add an auxiliary node to allow for SCALING DOWN test (Generative Surplus source 2)
    phoenix._nodes.append(Node("Aux-Node-1", role='aux', initial_load=0.1))
    phoenix._aux_node_counter = 1
    print("[SETUP] Added Aux-Node-1 manually for resource optimization test.")
```

```
    # 3. CLI Loop
    while True:
        display_menu(phoenix)

        try:
            choice = await asyncio.to_thread(input, "Enter command number: ")

            if choice == '1':
                # --- PROCESS INPUT (GENERATE LOAD/MOOD) ---
                user_input = await asyncio.to_thread(input, "Enter input data (use 'error', 'critical',
'good' to affect mood/risk): ")
                result = await phoenix.process_input({"data": user_input})
                print("\n[ACTION] Input Processed:")
                print(json.dumps(result, indent=2))

            elif choice == '2':
                # --- TRIGGER SELF-HEAL (HEALING SURPLUS SOURCE) ---
                print("\n[ACTION] Attempting Self-Healing...")
                # To ensure +0.05 surplus: manually set low aux load
                for n in phoenix._nodes:
                    if n.role == 'aux': n.load = 0.05
                    if n.mood < 0: n.mood = -2

                # Force state that triggers healing logic easily
                phoenix._current_mood_numeric = phoenix.config["MOOD_CRITICAL_LOW"] - 0.1
                phoenix._self_healing_active = True

                result = phoenix.trigger_heal()
                print(json.dumps(result, indent=2))

            elif choice == '3':
                # --- FORCE FAIL MIRROR (PRECURSOR TO STABILITY SURPLUS SOURCE) ---
                print("\n[ACTION] Forcing Mirror-2 offline...")
                result_msg = phoenix.fail_mirror(1) # Index 1 is Mirror-2
                print(f"[STATUS] {result_msg}")

            elif choice == '4':
                # --- TRIGGER REDUNDANCY REPAIR (STABILITY SURPLUS SOURCE) ---
                print("\n[ACTION] Attempting Redundancy Repair (Source of +0.01 Surplus)...")
                result = phoenix.trigger_redundancy_repair()
                print(json.dumps(result, indent=2))
```

```
        elif choice == '5':
            # --- ATTEMPT SCALING DOWN (RESOURCE OPTIMIZATION SURPLUS
SOURCE) ---
            print("\n[ACTION] Attempting Resource Optimization (Scaling Down)...")

            # To ensure +0.03 surplus: manually set low average load
            for n in phoenix._nodes: n.load = 0.05

            await phoenix._scaling_check_down()
            status = phoenix.get_status()
            print(f"[STATUS] Current Load: {status['total_load']:.3f}. Current Aux Nodes:
{status['scaling_status']['aux_count']}")

        elif choice == '6':
            # --- SHOW DETAILED STATUS ---
            status = phoenix.get_status()
            print("\n[STATUS] Detailed System Metrics:")
            print(json.dumps(status, indent=2))

        elif choice == '7':
            # --- EXPORT AUDIT LOG ---
            log = phoenix.export_audit_log(format="csv")
            print("\n--- GUARDIAN AUDIT LOG ---")
            print(log)
            print("--------------------------")


        elif choice == '8':
            print("\n[PHOENIX] Shutting down. Farewell, Operator.")
            sys.exit(0)

        else:
            print("Invalid command. Please enter a number from 1 to 8.")

    except Exception as e:
        print(f"\n[CRITICAL ERROR] An unexpected error occurred: {e}")

# --- ASYNC ENTRY POINT ---
if __name__ == "__main__":
    try:
        asyncio.run(main_cli())
    except KeyboardInterrupt:
        print("\n[PHOENIX] Shutdown by Operator.")
    except RuntimeError as e:
```

```
        # Allows running in environments where a loop is already running (e.g. jupyter, some IDEs)
        if "cannot run" in str(e):
            asyncio.get_event_loop().run_until_complete(main_cli())
        else:
            raise e
```

I felt this was as good as it was going to get all I wanted to do now was refine it a little more and eventually i got to 7.13 versions

And this is how the Final version looked like along with results -

```
# Phoenix System v7.13 - Critical Bug Fixes Applied
import time
import json
import random
import asyncio
import sys
import hashlib
import statistics
from datetime import datetime
from plyer import notification # V7.12: Desktop Notification Library

# --- CORE DATA STRUCTURES AND CONFIG (V7.13) ---

# Centralized System Configuration for Tunability
SYSTEM_CONFIG = {
    # Node Scaling Thresholds
    "LOAD_THRESHOLD_SCALE": 0.65,
    "LOW_LOAD_THRESHOLD": 0.20,
    "MAX_AUX_NODES": 10,
    "SCALING_UP_INCREMENT": 2,

    # Node Performance Metrics (Body/Mind)
    "LOAD_DECAY_FACTOR": 0.95,
    "LOAD_PASSIVE_DECAY": 0.01,
    "TASK_COST_BASE": 0.05,
    "TASK_COST_RANGE": 0.15,
    "BODY_EMERGENCY_THRESHOLD": 0.70,
    "SPIKE_LOAD_FACTOR": 0.25,

    # V7.7 Surplus Factors (Used in efficiency formula)
    "HEAL_SURPLUS_FACTOR": 1.5,
```

```
    "REPAIR_SURPLUS_FACTOR": 5.0,
    "SCALING_SURPLUS_FACTOR": 0.8,

    # V7.11: Surplus Costs and Recovery Multipliers
    "HEAL_COST": 0.2,
    "REPAIR_COST": 0.3,
    "SCALING_COST": 0.1,
    "FAST_RECOVERY_MULTIPLIER": 0.75,
    "SLOW_RECOVERY_MULTIPLIER": 0.4,

    # MoodNumeric Clamping & Healing Thresholds
    "MOOD_CRITICAL_LOW": -1.5,
    "MOOD_HEAL_TRIGGER": 0.5,
    "MOOD_CLAMP_MIN": -3,
    "MOOD_CLAMP_MAX": 3,

    # Guardian and Risk Thresholds
    "RISK_BLOCKED_THRESHOLD": 0.75,
    "RISK_WARNING_THRESHOLD": 0.50,
    "GUARDIAN_GRACE_TIMER": 5.0,

    # Soul Memory/Audit Configuration
    "SOUL_SNAPSHOT_THRESHOLD": 5,
    "SOUL_TREND_WINDOW": 10,

    # V7.10: Generative Surplus Safety Checks
    "MAX_SAFE_SURPLUS": 5.0,
    "GROWTH_RATE_LIMIT": 0.3,

    # V7.12: Notification Manager Configuration
    "SLEEP_MODE_ACTIVE": False, # Global flag for silencing non-critical alerts
    "BATCH_DELAY_SECONDS": 15, # Max time to wait for Tier 1/2 batching
    "NOTIFICATION_RISK_THRESHOLD": 0.5, # Risk factor that triggers a Tier 2 alert

    # Zero-Division Protection (v7.13)
    "MIN_LOAD_REDUCTION_EPSILON": 0.001,

    # Surplus Decay (v7.13)
    "SURPLUS_DECAY_ENABLED": True,
    "DAILY_DECAY_RATE": 0.05,
}

class PhoenixInput:
    """Input object with all optional fields explicitly defined."""
```

```
    def __init__(self, data: dict):
        self.timestamp = datetime.now().isoformat()
        self.data = data.get('data', 'No Data')
        self.type = data.get('type', 'text')
        self.priority = data.get('priority', 0.0)

        # V7.8: Explicitly named optional input fields
        self.radar = data.get('radar', None)
        self.coords = data.get('coords', None)
        self.binary = data.get('binary', None)
        self.audio = data.get('audio', None)
        self.image = data.get('image', None)
        self.iot = data.get('iot', None)

        # State tracking
        self.processed_by = None
        self.mood_impact = 0.0
        self.risk_factor = 0.0
        self.sandbox_confidence = 0.0

class Redundancy:
    """Redundancy mirror, tracks failure time for V7.7 Faith Surplus calculation."""
    def __init__(self, name: str):
        self.name = name
        self.is_active = True
        self.failure_time = None

    def simulate_failure(self):
        self.is_active = False
        self.failure_time = datetime.now()

class Node:
    """V7.11: Core or Auxiliary processing unit."""
    def __init__(self, name: str, role: str = 'aux', initial_load: float = 0.0, is_active: bool = True):
        self.name = name
        self.role = role
        self.load = initial_load
        self.current_load = initial_load
        self.mood = 0.0 # Initialized internally, can be set externally post-creation
        self.is_active = is_active

        self.is_failed_core = False
        self.core_role_backup = None
```

```
        self.last_processed_data = None
        self.mirror = Redundancy(f"Mirror-{name}")

    async def process(self, input_obj: PhoenixInput, body_load: float = 0.0, current_mood: float =
0.0) -> dict:
        """
        V7.11: Simulates task execution with Heart-Core mood adjusting Mind-Core delay.
        """
        if not self.is_active:
            return {"node": self.name, "result": "INACTIVE/FAILED", "processed_data": None}

        if self.role == 'observer':
            return {"node": self.name, "result": "Observer passively monitoring.", "processed_data":
None}

        processing_delay = 0.01
        result_text = "Task completed."

        if self.role == 'mind':
            task_complexity = len(input_obj.data.split()) / 10.0
            sub_tasks = max(2, int(task_complexity * 5))

            # V7.11: Dynamic Body/Mind Interaction (Mood-Adjusted Throttling)
            normalized_mood = max(0.0, min(1.0, (current_mood -
SYSTEM_CONFIG["MOOD_CLAMP_MIN"]) /
                                  (SYSTEM_CONFIG["MOOD_CLAMP_MAX"] -
SYSTEM_CONFIG["MOOD_CLAMP_MIN"])))

            mood_adjustment_factor = 1 + (1 - normalized_mood)

            body_stress_factor = (body_load ** 2) * mood_adjustment_factor
            processing_delay = 0.01 + (self.load * 0.05) + body_stress_factor

            result_text = (f"Input decomposed into {sub_tasks} micro-tasks. "
                           f"Prioritized critical tasks. (Body Stress Delay Factor: {processing_delay:.3f}s)")

        await asyncio.sleep(processing_delay)

        task_cost = SYSTEM_CONFIG["TASK_COST_BASE"] + random.random() *
SYSTEM_CONFIG["TASK_COST_RANGE"]
        self.load = min(1.0, self.load + task_cost)
        self.current_load = self.load
        self.mood += input_obj.mood_impact * 0.5
        self.last_processed_data = input_obj.data
```

```
        return {
            "node": self.name,
            "role": self.role,
            "result": f"{result_text} New Load: {self.load:.2f}, Mood: {self.mood:+.2f}",
            "processed_data": input_obj.data.upper()
        }


# --- GUARDIAN AND SANDBOX (V7.11) ---

class GuardianSystem:
    """V7.11: Guardian with continuous risk assessment."""
    def __init__(self, audit_log_callback, notification_manager, config):
        self.audit_log_callback = audit_log_callback
        self.notification_manager = notification_manager # V7.12: Inject manager
        self.config = config
        self.alert_level = 0
        self.last_neutralization_time = datetime.now()
        self.containment_active = False
        self._human_override_code = "1020"

    def watcher_scan(self, input_obj: PhoenixInput, system_nodes: list) -> float:
        """Watcher 👁️: Pattern Recognition. Sets base risk (0.0 - 1.0)."""
        risk = 0.0
        data_lower = input_obj.data.lower()

        # 1. Keyword Risk
        if 'override' in data_lower or 'exploit' in data_lower or 'threat' in data_lower:
            risk = max(risk, random.uniform(0.6, 0.8))

        # 2. Multi-Modal/Optional Input Risk
        optional_inputs = [input_obj.radar, input_obj.coords, input_obj.binary, input_obj.audio,
input_obj.image, input_obj.iot]
        if any(optional_inputs):
            risk = max(risk, 0.3)
            if input_obj.binary or input_obj.image:
                risk = max(risk, 0.6)
                self.audit_log_callback("WATCHER-PAYLOAD-ALERT", "High-risk binary/image
payload detected.", 0.0)
                # V7.12: Register Alert
                self.notification_manager.register_alert("Guardian: Payload Alert", "High-risk
binary/image payload detected in input.", 2)
```

```
        return risk

    def analyzer_assess(self, base_risk: float, sandbox_confidence: float) -> tuple[float, str]:
        """V7.11 Analyzer 🧱: Risk Assessment using continuous sandbox confidence."""

        final_risk = base_risk * sandbox_confidence
        analysis_detail = f"Initial Watcher Risk: {base_risk:.2f}. Sandbox Confidence:
{sandbox_confidence:.2f}."

        if final_risk >= self.config["RISK_BLOCKED_THRESHOLD"]:
            analysis_detail += " **HIGH RISK**: Confirmed high-threat signature."
            self.notification_manager.register_alert("Guardian: HIGH RISK", f"Confirmed threat
({final_risk:.2f}). Input will be blocked.", 2)
        elif final_risk >= self.config["RISK_WARNING_THRESHOLD"]:
            analysis_detail += " **WARNING**: Elevated risk due to pattern match/uncertainty."

        if final_risk > 1.0: final_risk = 1.0

        return final_risk, analysis_detail

    def containment_isolate(self, final_risk: float):
        """Containment 🛡️: Triggers system isolation based on risk."""
        is_active_before = self.containment_active
        if final_risk >= self.config["RISK_BLOCKED_THRESHOLD"]:
            self.containment_active = True
            if not is_active_before:
                # V7.12: Tier 3 Notification
                self.notification_manager.register_alert("GUARDIAN: CONTAINMENT ACTIVE",
"CRITICAL: Isolation protocols initiated due to threat.", 3)

        elif final_risk < self.config["RISK_WARNING_THRESHOLD"] and self.containment_active:
            self.containment_active = False
            self.notification_manager.register_alert("Guardian: Containment Dropped", "Isolation
protocols deactivated. System nominal.", 1)

    def neutralizer_execute(self, final_risk: float, system_mood: float, override_code: str = "") ->
bool:
        """Neutralizer ⚡: Shutdown with fully interactive Human Override check."""
        if final_risk >= self.config["RISK_BLOCKED_THRESHOLD"]:
            time_since_last_neutralization = (datetime.now() -
self.last_neutralization_time).total_seconds()

            if time_since_last_neutralization < self.config["GUARDIAN_GRACE_TIMER"]:
```

```
            self.audit_log_callback("NEUTRALIZER-GRACE", "Soft-shutdown skipped due to
Grace Timer.", 0.0)
            return False

        if system_mood < self.config["MOOD_CRITICAL_LOW"]:
            if override_code == self._human_override_code:
                self.audit_log_callback("NEUTRALIZER-HARD-OVERRIDE", f"Hard shutdown
authorized by two-person override. Risk: {final_risk:.2f}.", 0.0)
                self.last_neutralization_time = datetime.now()
                return True
            else:
                self.audit_log_callback("NEUTRALIZER-HARD-HOLD", "CRITICAL STATE:
System Frozen. Awaiting two-person override code (1020).", 0.0)
                # V7.12: Tier 3 Notification for Frozen State
                self.notification_manager.register_alert("NEUTRALIZER FROZEN", "CRITICAL:
System Frozen. Awaiting Human Override Code (1020).", 3)
                print("\n🚨 CRITICAL FAILURE 🚨: System is FROZEN. Awaiting Human Override
Code (1020).")
                return True

        self.audit_log_callback("NEUTRALIZER-SOFT", "Soft shutdown initiated. Awaiting
human action.", 0.0)
        self.last_neutralization_time = datetime.now()
        return True

    return False

class BabyPhoenixSandbox:
    """V7.11: Isolated Lab returns a continuous risk confidence score."""
    def __init__(self, config):
        self.config = config
        self.is_isolated = True
        self.last_input_hash = ""

    def pipe_input(self, input_obj: PhoenixInput) -> float:
        """V7.11: Pipes input for shadow-check and returns a continuous risk confidence score
(0.0-1.0)."""
        current_hash = hashlib.sha256(input_obj.data.encode('utf-8')).hexdigest()
        if hasattr(self, 'last_input_hash') and current_hash == self.last_input_hash:
            return 0.0
        self.last_input_hash = current_hash

        return self._evaluate_confidence(input_obj)
```

```
    def _evaluate_confidence(self, input_obj: PhoenixInput) -> float:
        """Calculates a confidence score (1.0 = highly confident it's a threat)."""
        confidence = 0.0

        data_lower = input_obj.data.lower()
        if 'exploit' in data_lower or 'critical_formula_breach' in data_lower:
            confidence = max(confidence, random.uniform(0.85, 1.0))
        elif 'caution' in data_lower or 'threat' in data_lower:
            confidence = max(confidence, random.uniform(0.5, 0.75))

        has_high_impact_payload = input_obj.binary or input_obj.image
        if has_high_impact_payload:
            confidence = max(confidence, random.uniform(0.7, 0.9))

        has_low_impact_inputs = input_obj.radar or input_obj.coords or input_obj.audio or
input_obj.iot
        if has_low_impact_inputs:
            confidence = max(confidence, random.uniform(0.3, 0.5))

        return min(1.0, confidence)


# --- NOTIFICATION MANAGER (V7.12) ---

class NotificationManager:
    """
    V7.12: Centralized Gatekeeper for desktop notifications.
    Handles queueing, tier evaluation, batching, and logging.
    """
    def __init__(self, phoenix_system):
        self.phoenix = phoenix_system
        self.queue = []
        self.last_batch_time = time.time()

    def register_alert(self, title: str, message: str, tier: int = 2):
        """Registers an event to the queue for later processing."""
        self.queue.append({'title': title, 'message': message, 'tier': tier, 'timestamp': time.time()})

    def process_queue(self):
        """Evaluates the queue, sends critical alerts immediately, and batches others."""
        if not self.queue:
            return

        tier3 = [e for e in self.queue if e['tier'] == 3]
```

```
        tier2 = [e for e in self.queue if e['tier'] == 2]
        tier1 = [e for e in self.queue if e['tier'] == 1]

        current_time = time.time()
        sleep_mode = self.phoenix.config["SLEEP_MODE_ACTIVE"]

        # 1. Send Tier 3 immediately (Always breaks sleep mode)
        for event in tier3:
            self._send(event)

        # 2. Process Tier 1/2 based on batching and sleep mode
        if current_time - self.last_batch_time >= self.phoenix.config["BATCH_DELAY_SECONDS"]
or tier2:
            self.last_batch_time = current_time # Reset timer after sending a batch

            # Process Tier 2 Alerts
            if tier2:
                if not sleep_mode:
                    # Send Tier 2 as a combined warning
                    combined_msg = "\n".join([f"[{e['title']}] {e['message']}" for e in tier2])
                    self._send({'title': f"Phoenix WARNING [Tier 2]", 'message': combined_msg, 'tier':
2})
                else:
                    self.phoenix._log_audit_entry("NOTIFICATION-HELD", f"Tier 2 alerts held due to
Sleep Mode: {len(tier2)} alerts.", 0.0)

            # Process Tier 1 Alerts (Only if not in sleep mode)
            if tier1 and not sleep_mode:
                combined_msg = "\n".join([f"[{e['title']}] {e['message']}" for e in tier1])
                self._send({'title': f"Phoenix INFO [Tier 1]", 'message': combined_msg, 'tier': 1})
            elif tier1 and sleep_mode:
                 self.phoenix._log_audit_entry("NOTIFICATION-SUPPRESSED", f"Tier 1 alerts
suppressed due to Sleep Mode: {len(tier1)} alerts.", 0.0)

        # Clear sent alerts (Tier 3 is always cleared; Tier 1/2 are cleared if processed/batched)
        self.queue = [e for e in self.queue if e['tier'] == 3] # Keep Tier 1/2 in queue until batching
time hits

    def _send(self, event):
        """Sends the notification and logs the action."""

        # Determine Title prefix and Sound (using default OS sounds)
        prefix = {3: "🚨 CRITICAL 🚨", 2: "⚠️ WARNING ⚠️", 1: "💡 Info"}.get(event['tier'])
        title = f"{prefix} | {event['title']}"
```

```
        try:
            notification.notify(
                title=title,
                message=event['message'],
                timeout=10,
                # Optional: use toast for Windows or other platform-specific features
                app_name="Phoenix V7.13"
            )
        except Exception as e:
            # Fallback for environments where plyer can't create a desktop notification
            print(f"[ERROR] Could not send desktop notification: {e}")

        # Log to Audit (Tier 3 is logged immediately, Tier 1/2 are logged when batched)
        log_detail = f"Tier {event['tier']} | {event['title']}: {event['message']}"
        self.phoenix._log_audit_entry("NOTIFICATION-SENT", log_detail, 0.0)


# --- PHOENIX SYSTEM CORE (V7.13) ---

class PhoenixSystem:
    """
    V7.13: The Fusion Core with critical bug fixes for zero-division and surplus decay.
    """

    def __init__(self, name: str = "Phoenix System v7.13", initial_nodes: list = None):
        self.config = SYSTEM_CONFIG
        self.name = name

        self._nodes = []
        for n_data in (initial_nodes if initial_nodes is not None else []):
            initial_mood = n_data.pop('mood', 0.0)
            new_node = Node(is_active=True, **n_data)
            new_node.mood = initial_mood
            self._nodes.append(new_node)

        # V7.12: Initialize Notification Manager before Guardian
        self.notification_manager = NotificationManager(self)

        self._guardian = GuardianSystem(self._log_audit_entry, self.notification_manager,
self.config)
        self._sandbox = BabyPhoenixSandbox(self.config)

        self._aux_node_counter = 0
```

```
        # State & Memory
        self._soul_memory = []
        self._audit_log = []
        self._soul_snapshots = []
        self._mood_history = [0.0] * self.config["SOUL_TREND_WINDOW"]
        self._generative_surplus = 0.0
        self._current_mood_numeric = 1.0
        self._self_healing_active = False
        self._dual_location_storage_status = "Online (Simulated Secure Vault)"
        self._latest_sandbox_confidence = 0.0

        # V7.10: Surplus Safety Tracking
        self._last_surplus_value = 0.0
        self._last_surplus_check_time = time.time()

        print(f"\n[INIT] {self.name} Core initialized. Guardian, Sandbox, and Notification Manager
online.")

    def _apply_surplus_decay(self):
        """v7.13: Apply daily surplus decay to prevent infinite accumulation"""
        if not self.config.get("SURPLUS_DECAY_ENABLED", True):
            return

        now = datetime.now()
        if not hasattr(self, '_last_surplus_decay_time'):
            self._last_surplus_decay_time = now
            return

        hours_passed = (now - self._last_surplus_decay_time).total_seconds() / 3600

        if hours_passed >= 24:  # Apply decay every 24 hours
            if self._generative_surplus > 0:
                old_surplus = self._generative_surplus
                self._generative_surplus *= (1 - self.config["DAILY_DECAY_RATE"])
                decayed = old_surplus - self._generative_surplus
                if decayed > 0.01:
                    self._log_audit_entry("SURPLUS-DECAY", f"Applied decay: {old_surplus:.4f} →
{self._generative_surplus:.4f} (-{decayed:.4f})", -decayed)
            self._last_surplus_decay_time = now

    # --- SOUL TREND ANALYSIS & INTEGRITY (V7.8) ---

    def _shadow_soul_validation(self):
```

```
        """
        V7.11: Shadow Check requires low sandbox confidence for integrity pass.
        """
        if self._latest_sandbox_confidence >= self.config["RISK_BLOCKED_THRESHOLD"]:
            self._log_audit_entry("SOUL-BREACH", "CRITICAL: Sandbox confidence too high.
Write integrity suspect.", 0.0)
            self.notification_manager.register_alert("Soul Integrity Breach", "Sandbox confidence is
high during soul write. Integrity suspect.", 2)
            return False

        if self._soul_snapshots and random.random() > 0.99:
            self._log_audit_entry("SOUL-HARDWARE-FAIL", "CRITICAL: Shadow Soul Validation
failed (Hardware). Integrity breach suspected.", 0.0)
            self.notification_manager.register_alert("Soul: Hardware Failure", "CRITICAL: Shadow
Soul Validation failed (Hardware).", 3)
            return False

        return True

    def _calculate_trend_analysis(self) -> str:
        """Soul 🌌: Expands long-term memory analysis to track behavioral trends."""
        if len(self._mood_history) < 2:
            return "Insufficient data."

        recent_moods = self._mood_history
        trend_window = self.config["SOUL_TREND_WINDOW"]
        avg_recent = statistics.mean(recent_moods[-trend_window // 2:])
        avg_long = statistics.mean(recent_moods)

        if avg_recent < avg_long - 0.5:
            # V7.12: Register Alert for significant decline
            self.notification_manager.register_alert("Mood Trend Alert", "Significant recent decline in
mood trend detected.", 2)
            return "Significant recent decline (MOOD STRESS WARNING)."
        elif self._current_mood_numeric < 0 and avg_long < 0:
            return "Sustained negative drift (RISK OF ETHICAL DEVIATION)."
        return "Mood trend stable."

    def _log_audit_entry(self, event: str, detail: str, surplus: float = 0.0):
        """Audit 📜: Centralized logging for the Guardian Audit Log (Immutable Ledger)."""
        self._audit_log.append({
            "timestamp": datetime.now().isoformat(),
            "event": event,
            "detail": detail,
```

```
            "surplus": surplus,
            "system_mood_numeric": f"{self._current_mood_numeric:+.2f}",
            "soul_trend_summary": self._calculate_trend_analysis()
        })

    def _create_soul_snapshot(self):
        """Creates a rolling cryptographic snapshot."""
        if len(self._soul_memory) >= self.config["SOUL_SNAPSHOT_THRESHOLD"]:
            soul_data_json = json.dumps(self._soul_memory, sort_keys=True)
            snapshot_hash = hashlib.sha256(soul_data_json.encode('utf-8')).hexdigest()

            location_a_time = (datetime.now() - datetime.fromtimestamp(time.time() -
random.uniform(0.01, 0.05))).isoformat()
            location_b_time = (datetime.now() - datetime.fromtimestamp(time.time() -
random.uniform(0.06, 0.10))).isoformat()

            self._soul_snapshots.append({
                "timestamp": datetime.now().isoformat(),
                "hash": snapshot_hash,
                "dual_location_write_A": location_a_time,
                "dual_location_write_B": location_b_time
            })
            self._soul_memory = []

            self._shadow_soul_validation()
            self._log_audit_entry("SOUL-SNAPSHOT", f"Immutable snapshot created. Dual-location
sync successful.", 0.0)

    # --- REDUNDANCY AND FAILOVER (V7.7) ---

    def _core_node_failover_check(self):
        """Checks for CORE node failure and initiates active load transfer."""
        core_roles = ['heart', 'mind', 'body']

        for core_role in core_roles:
            core_node = next((n for n in self._nodes if n.role == core_role and n.is_active), None)

            if core_node and not core_node.mirror.is_active:
                core_node.is_active = False
                self._log_audit_entry("CORE-FAILOVER-TRIGGER", f"CRITICAL: Core {core_role}
mirror failed. Initiating load transfer.", 0.0)

                available_aux = sorted([
                    n for n in self._nodes
```

```
                if n.role == 'aux' and n.is_active and not n.is_failed_core
            ], key=lambda x: x.load)

            if available_aux:
                backup_node = available_aux[0]

                backup_node.role = core_role
                backup_node.is_failed_core = True
                backup_node.core_role_backup = core_node.name

                backup_node.load = max(backup_node.load, core_node.load)
                backup_node.mood = core_node.mood

                self._log_audit_entry("CORE-FAILOVER-COMPLETE", f"Load transferred from
{core_node.name} to {backup_node.name} (now acting as {core_role}).", 0.0)
                # V7.12: Tier 3 Notification for Core Failover
                self.notification_manager.register_alert("CORE FAILOVER", f"CRITICAL:
{core_role} failed. Load transferred to {backup_node.name}.", 3)
            else:
                self._log_audit_entry("CORE-FAILOVER-FAILED", f"CRITICAL: No available Aux
node to back up {core_role}.", 0.0)
                # V7.12: Tier 3 Notification for Failover Failure
                self.notification_manager.register_alert("CORE FAILOVER FAILURE", f"CRITICAL:
No Aux node available to back up {core_role}!", 3)

    def _cross_node_shadow_check(self):
        """Simulates cross-node failover/integrity checks across all active nodes."""
        failed_mirrors = [n.name for n in self._nodes if not n.mirror.is_active and n.role == 'aux']
        if failed_mirrors:
            self._log_audit_entry("AUX-SHADOW-ALERT", f"ALERT: {len(failed_mirrors)} Aux
mirrors failed. Repair protocols ongoing.", 0.0)
            self.notification_manager.register_alert("Aux Mirror Alert", f"{len(failed_mirrors)} Aux
mirrors failed. Repair protocols active.", 2)

            for node in self._nodes:
                if node.role == 'aux' and not node.mirror.is_active and random.random() < 0.5:
                    node.mirror.is_active = True
                    self._log_audit_entry("AUX-HEAL", f"{node.name} successfully repaired its own
mirror.", 0.0)

    # --- GENERATIVE SURPLUS & SPENDING (V7.13) ---

    def _enforce_surplus_safety(self, generated_amount: float, action: str):
        """
```

```
        V7.10: Enforces maximum surplus and maximum growth rate checks via assert.
        """

        # 1. Growth Rate Check (Enforced on generated_amount per operation)
        assert generated_amount <= self.config["GROWTH_RATE_LIMIT"], (
            f"[{action} - ASSERT] Surplus growth too fast! Generated {generated_amount:.4f}
exceeds limit {self.config['GROWTH_RATE_LIMIT']:.4f}."
        )

        # 2. Max Surplus Check (Potential total)
        potential_surplus = self._generative_surplus + generated_amount
        if potential_surplus > self.config["MAX_SAFE_SURPLUS"]:
            # V7.12: Tier 2 Notification
            self.notification_manager.register_alert("Surplus Overload", f"Potential surplus
({potential_surplus:.2f}) exceeds max safe level.", 2)

        self._generative_surplus = potential_surplus

        self._last_surplus_value = self._generative_surplus
        self._last_surplus_check_time = time.time()

    def _spend_surplus(self, required_cost: float, action: str) -> bool:
        """V7.11: Attempts to spend surplus resource to offset operational cost."""
        if self._generative_surplus >= required_cost:
            self._generative_surplus -= required_cost
            self._log_audit_entry("SURPLUS-SPENT",
                                  f"Used {required_cost:.4f} surplus for {action}. Remaining:
{self._generative_surplus:.4f}.",
                                  -required_cost)
            return True
        return False

    def _generate_suggestion(self, input_obj: PhoenixInput) -> str:
        """Suggestion Box with tight integration to optional input context."""
        mood = self._current_mood_numeric
        rec = ""

        if mood < self.config["MOOD_CRITICAL_LOW"]:
            rec += "CRITICAL: Recommend immediate **Human Override** (Code 1020) and Hard
Throttling."
        elif mood < self.config["MOOD_HEAL_TRIGGER"]:
            can_afford_heal = self._generative_surplus >= self.config["HEAL_COST"]
            afford_status = "FAST " if can_afford_heal else "STANDARD "
```

```
            rec += f"PRIORITY: Mood is low ({mood:+.2f}). Execute **Self-Healing** (Option 2) for {afford_status}recovery."
        elif mood > 2.0 and self._get_total_load() < 0.2 and self._aux_node_counter > 0:
            rec += "OPTIMIZATION: Initiate **Aux Node Scaling Down** (Option 5) for Will Surplus."
        else:
            rec += "STATUS QUO: Monitor operations. Repair FAILED Aux mirrors (Option 4) for Faith Surplus."

        if input_obj.binary is not None or input_obj.image is not None:
            rec += " **SECURITY ALERT**: High-impact payload detected. **Mandatory** deep-scan protocol initiation recommended."
        if self._guardian.containment_active:
             rec += " **CRITICAL**: Guardian Containment is ACTIVE. Review audit log immediately."

        return rec

    # --- WORKFLOW EXECUTION ---

    async def process_input(self, input_data: dict, override_code: str = "") -> dict:
        """Executes the full Blueprint Workflow (V7.13)."""

        input_obj = PhoenixInput(input_data)

        self._apply_surplus_decay()  # v7.13: Apply surplus decay before processing

        total_load = self._get_total_load()
        if total_load < self.config["LOAD_THRESHOLD_SCALE"] / 2.0 and random.random() < 0.3:
            spike_load = random.uniform(0.01, self.config["SPIKE_LOAD_FACTOR"])
            for n in self._nodes:
                if n.role in ('heart', 'mind', 'body') and n.is_active:
                    n.load = min(1.0, n.load + spike_load)
            self._log_audit_entry("SPIKE-LOAD-TEST", f"Simulated a load spike of {spike_load:.2f} across core nodes.", 0.0)
            self.notification_manager.register_alert("Load Spike Test", f"Simulated load spike of {spike_load:.2f} across core nodes.", 1)

        # 1. Watcher Scan
        base_risk = self._guardian.watcher_scan(input_obj, self._nodes)

        # 2. Sandbox Testing
        input_obj.risk_factor = base_risk
        sandbox_confidence = self._sandbox.pipe_input(input_obj)
        input_obj.sandbox_confidence = sandbox_confidence
```

```
        self._latest_sandbox_confidence = sandbox_confidence

        # 3. Guardian Analyzer/Neutralizer Check
        final_risk, analysis_detail = self._guardian.analyzer_assess(base_risk,
sandbox_confidence)

        if final_risk >= self.config["NOTIFICATION_RISK_THRESHOLD"]:
            self.notification_manager.register_alert("Risk Alert", f"Input risk is {final_risk:.2f}. See
analysis.", 2)

        input_obj.mood_impact = (final_risk * -3) + random.uniform(-0.5, 0.5)
        self._guardian.containment_isolate(final_risk)

        if self._guardian.neutralizer_execute(final_risk, self._current_mood_numeric,
override_code):
            if self._current_mood_numeric < self.config["MOOD_CRITICAL_LOW"] and
override_code != self._guardian._human_override_code:
                # Tier 3 notification is already registered in neutralizer_execute
                return {"status": "GUARDIAN-FROZEN", "message": f"CRITICAL THREAT. System is
FROZEN, requires Human Override Code: {analysis_detail}"}
            else:
                return {"status": "GUARDIAN-SHUTDOWN", "message": f"CRITICAL THREAT.
Neutralizer executed: {analysis_detail}"}

        # 4. Input Blocking
        if final_risk >= self.config["RISK_BLOCKED_THRESHOLD"]:
            self._log_audit_entry("GUARDIAN-BLOCK", f"Input blocked due to high risk
({final_risk:.2f}).", 0.0)
            return {"status": "BLOCKED by Guardian V7.13", "message": "Input blocked by
Neutralizer/Containment.", "risk_factor": final_risk}

        # 5. Core Processing
        body_node = next((n for n in self._nodes if n.role == 'body' and n.is_active), None)
        body_load_for_mind = body_node.current_load if body_node else 0.0

        processing_tasks = [
            n.process(input_obj,
                      body_load=body_load_for_mind,
                      current_mood=self._current_mood_numeric)
            for n in self._nodes if n.role != 'observer' and n.is_active
        ]
        results = await asyncio.gather(*processing_tasks)
        execution_result = results[0] if results else {"node_output": "N/A"}
```

```
    # 6. Core Monitoring and Integrity Checks (Post-Processing)
    self._update_global_mood()
    self._create_soul_snapshot()
    self._core_node_failover_check()
    self._cross_node_shadow_check()
    await self._scaling_check()
    await self._scaling_check_down()

    # V7.12: Process Notification Queue at the end of the loop
    self.notification_manager.process_queue()

    self._soul_memory.append({
        "timestamp": input_obj.timestamp,
        "input_preview": input_obj.data[:30],
        "risk": final_risk,
        "sandbox_confidence": input_obj.sandbox_confidence
    })

    mood_label, _ = self._get_mood_suggestion()
    suggestion_box_output = self._generate_suggestion(input_obj)

    return {
        "status": "Processed",
        "node_output": execution_result,
        "system_mood_numeric": f"{self._current_mood_numeric:+.2f}",
        "mood_label": mood_label,
        "guardian_analysis": analysis_detail,
        "sandbox_confidence": f"{input_obj.sandbox_confidence:.3f}",
        "soul_trend": self._calculate_trend_analysis(),
        "suggestion_box": suggestion_box_output
    }

# --- UTILITY AND MAINTENANCE ---

def _decay_load(self):
    """Decays load on all nodes, with Body-induced throttling on Aux nodes."""
    body_node = next((n for n in self._nodes if n.role == 'body' and n.is_active), None)
    body_load_stress = body_node.load if body_node else 0.0

    for node in self._nodes:
        if not node.is_active: continue

        decay_factor = self.config["LOAD_DECAY_FACTOR"]
        fixed_decay = self.config["LOAD_PASSIVE_DECAY"]
```

```
            if node.role.startswith('aux') and body_load_stress >
self.config["BODY_EMERGENCY_THRESHOLD"]:
                decay_factor *= 0.9
                fixed_decay *= 2.0
                self._log_audit_entry("BODY-THROTTLE", f"Emergency throttling on Aux node
{node.name}.", 0.0)
                # V7.12: Tier 2 Alert
                self.notification_manager.register_alert("Body Throttle", f"Emergency throttling on Aux
node {node.name}.", 2)


            node.load = max(0.0, node.load * decay_factor - fixed_decay)
            node.current_load = node.load

    def _get_total_load(self) -> float:
        """Calculates total load only across active, processing nodes."""
        self._decay_load()
        active_nodes_load = [n.load for n in self._nodes if n.role not in ('observer') and n.is_active]
        return sum(active_nodes_load) / len(active_nodes_load) if active_nodes_load else 0.0

    def _observer_clamp_mood(self):
        """Observer 👁️: Clamps all Node Moods to the Blueprint range (±3)."""
        for node in self._nodes:
            original_mood = node.mood
            node.mood = max(self.config["MOOD_CLAMP_MIN"],
min(self.config["MOOD_CLAMP_MAX"], node.mood))
            if original_mood != node.mood:
                self._log_audit_entry("OBSERVER-CLAMP", f"{node.name} clamped from
{original_mood:+.2f} to {node.mood:+.2f}.", 0.0)

    def _update_global_mood(self):
        """Updates global MoodNumeric based on active nodes."""
        self._observer_clamp_mood()
        active_moods = [n.mood for n in self._nodes if n.role not in ('observer') and n.is_active]
        if not active_moods:
            self._current_mood_numeric = 0.0
            return

        new_mood = sum(active_moods) / len(active_moods)
        self._current_mood_numeric = new_mood
        self._self_healing_active = new_mood < self.config["MOOD_HEAL_TRIGGER"]

        self._mood_history.append(new_mood)
```

```
        if len(self._mood_history) > self.config["SOUL_TREND_WINDOW"]:
            self._mood_history.pop(0)

        if self._current_mood_numeric < self.config["MOOD_CRITICAL_LOW"]:
            self.notification_manager.register_alert("Mood Critical", f"System MoodNumeric is
{self._current_mood_numeric:+.2f}.", 3)
        elif self._current_mood_numeric < 0:
            self.notification_manager.register_alert("Mood Stress", f"System MoodNumeric is
negative: {self._current_mood_numeric:+.2f}.", 2)

    def _get_mood_suggestion(self) -> tuple[str, str]:
        """Simple mood description based on current MoodNumeric."""
        mood = self._current_mood_numeric
        if mood > 2.0: return "Joyous (Critical Surplus)", "Max operational readiness."
        if mood > 1.0: return "Stable (High Performance)", "Normal operation."
        if mood > 0.0: return "Content (Optimal)", "Normal operation."
        if mood > -1.0: return "Neutral (Monitoring)", "Low-level stress detected."
        if mood > -2.0: return "Stressed (Self-Heal Required)", "Proactive stabilization needed."
        return "Critical (Guardian Alert)", "Immediate intervention required."

    # --- SCALING AND SURPLUS GENERATION (V7.13) ---

    def trigger_heal(self) -> dict:
        """Triggers mood/load stabilization and calculates **Love Surplus** based on efficiency."""
        initial_mood = self._current_mood_numeric

        if initial_mood > self.config["MOOD_HEAL_TRIGGER"]:
            return {"status": "Skipped", "message": "Mood is stable. Healing unnecessary."}

        has_spent_surplus = self._spend_surplus(self.config["HEAL_COST"], "Self-Heal")
        load_reduction_factor = self.config["FAST_RECOVERY_MULTIPLIER"] if
has_spent_surplus else self.config["SLOW_RECOVERY_MULTIPLIER"]
        load_reduced_amount = 0.0

        for node in self._nodes:
            if node.role.startswith('aux') and node.is_active:
                reduction = node.load * load_reduction_factor
                node.load = max(0.0, node.load - reduction)
                load_reduced_amount += reduction
            if node.mood < 0:
                node.mood = min(0.0, node.mood + 1.0)

        self._update_global_mood()
        final_mood = self._current_mood_numeric
```

```
        mood_improvement = final_mood - initial_mood

        surplus_generated = 0.0
        if load_reduced_amount > 0 and mood_improvement > 0.01:
            load_reduced_amount = max(load_reduced_amount,
self.config["MIN_LOAD_REDUCTION_EPSILON"])  # v7.13: Zero-division fix
            surplus_generated = (mood_improvement / load_reduced_amount) *
self.config["HEAL_SURPLUS_FACTOR"]
            surplus_generated = min(surplus_generated, self.config["GROWTH_RATE_LIMIT"] -
0.01)
        elif load_reduced_amount == 0 and mood_improvement > 0.01:
             surplus_generated = 0.08

        if surplus_generated > 0:
            self._enforce_surplus_safety(surplus_generated, "LOVE-HEAL")
            self._log_audit_entry("SELF-HEAL-EFFICIENT", f"Mood stabilized. Love Surplus:
{surplus_generated:.4f}.", surplus_generated)
            # V7.12: Tier 1 Notification for Surplus Generation
            self.notification_manager.register_alert("Surplus Generation (Love)",
f"+{surplus_generated:.4f} generated by efficient self-heal.", 1)

        else:
            self._log_audit_entry("SELF-HEAL-INEFFICIENT", "Healing complete, but
improvement/cost ratio too low for surplus.", 0.0)

        return {"status": "Complete", "message": f"Stabilized loads and mood. Mood improved by
{mood_improvement:+.2f}.", "avg_mood": final_mood, "surplus_spent": has_spent_surplus}

    def trigger_redundancy_repair(self, node_name: str) -> dict:
        """Repairs a failed redundancy mirror, calculating **Faith Surplus** based on downtime."""
        node = next((n for n in self._nodes if n.name == node_name), None)
        mirror = node.mirror if node else None

        if not mirror or mirror.is_active:
            return {"status": "Skipped", "message": f"Mirror for {node_name} is currently active or not
found."}

        has_spent_surplus = self._spend_surplus(self.config["REPAIR_COST"],
"Redundancy-Repair")

        time_since_failure = max((datetime.now() - mirror.failure_time).total_seconds() if
mirror.failure_time else 1.0, 1.0)  # v7.13: Zero-division fix

        surplus_generated = 0.0
```

```
        repair_status = "Skipped"

        if has_spent_surplus or random.random() < 0.9:
            mirror.is_active = True
            repair_status = "Successfully brought back online (Expedited)." if has_spent_surplus
else "Successfully brought back online (Standard)."

            surplus_generated = (1 / time_since_failure) *
self.config["REPAIR_SURPLUS_FACTOR"]
            surplus_generated = min(surplus_generated, self.config["GROWTH_RATE_LIMIT"] -
0.01)
        else:
            repair_status = "Repair failed: Requires additional resources or time."
            self.notification_manager.register_alert("Redundancy Repair Failed", f"Repair of
{mirror.name} failed. Retesting recommended.", 2)

        if node.is_failed_core and mirror.is_active:
            # Core restoration logic
            node.role = node.core_role_backup
            node.is_failed_core = False
            node.core_role_backup = None
            node.is_active = True
            self._log_audit_entry("CORE-RECOVERY", f"Original Core {node_name} recovered and
restored its role.", 0.0)
            self.notification_manager.register_alert("CORE RESTORED", f"Original {node_name}
recovered and restored its core role.", 1)

        if surplus_generated > 0:
            self._enforce_surplus_safety(surplus_generated, "FAITH-REPAIR")
            self._log_audit_entry("REDUNDANCY-REPAIRED", f"Repaired {mirror.name}. Faith
Surplus: {surplus_generated:.4f}.", surplus_generated)
            # V7.12: Tier 1 Notification for Surplus Generation
            self.notification_manager.register_alert("Surplus Generation (Faith)",
f"+{surplus_generated:.4f} generated by redundancy repair.", 1)

        return {"status": repair_status, "message": f"{mirror.name} status: {repair_status}",
"surplus_spent": has_spent_surplus}

    async def _scaling_check(self):
        """Dynamically scales up auxiliary nodes aggressively if load is high."""
        if self._get_total_load() > self.config["LOAD_THRESHOLD_SCALE"]:
            nodes_to_add = self.config["SCALING_UP_INCREMENT"]

            for _ in range(nodes_to_add):
```

```
            if self._aux_node_counter < self.config["MAX_AUX_NODES"]:
                self._aux_node_counter += 1
                new_node_name = f"Aux-Node-{self._aux_node_counter}"
                self._nodes.append(Node(new_node_name, role='aux', initial_load=0.1,
is_active=True))
                self._log_audit_entry("SCALING-UP", f"Scaled up {new_node_name}.", 0.0)
                # V7.12: Tier 1 Notification
                self.notification_manager.register_alert("Scaling UP", f"New Aux Node added:
{new_node_name}.", 1)

    async def _scaling_check_down(self):
        """Scales down and calculates **Will Surplus** based on the shed node's load cost."""
        current_load = self._get_total_load()
        if current_load < self.config["LOW_LOAD_THRESHOLD"] and self._aux_node_counter >
0:

            node_to_remove = next((n for n in reversed(self._nodes) if n.role == 'aux' and not
n.is_failed_core), None)

            if node_to_remove:
                self._spend_surplus(self.config["SCALING_COST"], "Scaling-Down")

                surplus_generated = node_to_remove.load *
self.config["SCALING_SURPLUS_FACTOR"]

                self._nodes.remove(node_to_remove)
                self._aux_node_counter -= 1

                self._enforce_surplus_safety(surplus_generated, "WILL-SCALING-DOWN")

                self._log_audit_entry("SCALING-DOWN-OPTIMIZED", f"Shed
{node_to_remove.name}. Will Surplus: {surplus_generated:.4f}.", surplus_generated)
                # V7.12: Tier 1 Notification
                self.notification_manager.register_alert("Scaling DOWN", f"Shed Aux Node:
{node_to_remove.name}. Will Surplus generated.", 1)

    def fail_mirror(self, node_name: str) -> str:
        """Utility to manually fail a mirror for testing repair logic."""
        node = next((n for n in self._nodes if n.name == node_name), None)
        mirror = node.mirror if node else None

        if mirror and mirror.is_active:
            mirror.simulate_failure()
            self._log_audit_entry("MIRROR-FAIL", f"{mirror.name} manually failed.", 0.0)
```

```
            self.notification_manager.register_alert("Mirror Failure", f"{mirror.name} manually failed.",
2)
            return f"{mirror.name} has been manually failed."
        return f"Mirror for {node_name} is already failed/offline or not found."

    def get_status(self) -> dict:
        """Returns a comprehensive status report of the Phoenix System."""
        mood_label, _ = self._get_mood_suggestion()
        node_loads = []
        for n in self._nodes:
            status_label = "ACTIVE" if n.is_active else "FAILED"
            role_label = n.role
            if n.is_failed_core:
                role_label = f"AUX (BACKING UP {n.core_role_backup.split('-')[0].upper()})"
            elif n.role in ('heart', 'mind', 'body') and not n.is_active:
                role_label = f"CORE (FAILED - BACKED UP)"

            node_loads.append({n.name: {
                "role": role_label,
                "load": f"{n.current_load:.2f}",
                "mood": f"{n.mood:+.2f}",
                "physical_status": status_label,
                "mirror": "Active" if n.mirror.is_active else "FAILED"
            }})

        return {
            "system_name": self.name,
            "avg_mood": f"{self._current_mood_numeric:+.2f}",
            "mood_label": mood_label,
            "total_load": self._get_total_load(),
            "guardian_containment_active": self._guardian.containment_active,
            "generative_surplus": self._generative_surplus,
            "scaling_status": {"aux_count": self._aux_node_counter},
            "sleep_mode": self.config["SLEEP_MODE_ACTIVE"], # V7.12: Add Sleep Mode Status
            "soul_memory_snapshot": {"snapshots": len(self._soul_snapshots),
"dual_location_status": self._dual_location_storage_status},
            "soul_trend_analysis": self._calculate_trend_analysis(),
            "node_loads": node_loads
        }

    def export_audit_log(self, format: str = "csv") -> str:
        """Exports the Guardian Audit Log (Athena's immutable ledger)."""
        if not self._audit_log:
            return "Audit log is empty."
```

```
        if format == "csv":
            header = "Timestamp,Event,Detail,Surplus,SystemMoodNumeric,SoulTrendSummary\n"
            rows = []
            for entry in self._audit_log:
                detail = str(entry.get('detail', 'N/A')).replace(",", ";")
                surplus_val = entry.get('surplus', 0.0)
                mood_val = entry.get('system_mood_numeric', 'N/A')
                trend_val = entry.get('soul_trend_summary', 'N/A').replace(",", ";")

rows.append(f"{entry['timestamp']},{entry['event']},{detail},{surplus_val:.4f},{mood_val},{trend_val
}")
            return header + "\n".join(rows)

        return json.dumps(self._audit_log, indent=2)

# --- CLI INTERFACE LOGIC ---

def display_menu(phoenix: PhoenixSystem):
    status = phoenix.get_status()
    sleep_status = "ACTIVE" if phoenix.config["SLEEP_MODE_ACTIVE"] else "DORMANT"

    print("\n" + "💰"*20 + " SHIN PHOENIX V7.13 (CRITICAL BUG FIXES) " + "💰"*20)
    print(f"| Load Status: {status['mood_label']} | MoodNumeric: {status['avg_mood']} | CORE
FAILOVER: {'ACTIVE' if 'FAILED' in status['node_loads'][0].get('Heart-Core',
{}).get('physical_status', 'ACTIVE') else 'NOMINAL'}")
    print(f"| Aux Nodes: {status['scaling_status']['aux_count']} | Total Load: {status['total_load']:.3f}
| Surplus: {status['generative_surplus']:.4f} | SLEEP MODE: {sleep_status}")
    print("-" * 92)
    print("1. Process Input (Test Dynamic Delay & Granular Risk)")
    print("2. Trigger Self-Heal (Test **Love Surplus** / Tier 1 Notification)")
    print("3. Fail CORE Node Mirror (Test Tier 2/3 Notification)")
    print("4. Repair CORE Mirror (Test **Faith Surplus** / Tier 1 Notification)")
    print("5. Attempt Scaling Down (Test **Will Surplus** / Tier 1 Notification)")
    print("6. Show Detailed Status")
    print("7. Export Audit Log (Includes Notification Logs)")
    print("8. Toggle Sleep Mode (Currently: {s})".format(s=sleep_status))
    print("9. Trigger Critical Shutdown (Tests Tier 3 Notification & Override)")
    print("0. Exit")
    print("="*92)

async def main_cli():
    # Corrected initial nodes definition:
    initial_nodes = [
```

```
        {"name": "Heart-Core", "role": "heart", "initial_load": 0.05, "mood": 1.0},
        {"name": "Mind-Core", "role": "mind", "initial_load": 0.1},
        {"name": "Body-Core", "role": "body", "initial_load": 0.15},
        {"name": "Observer-Meta", "role": "observer", "initial_load": 0.0}
    ]
    phoenix = PhoenixSystem(name="Shin Phoenix V7.13 (Critical Bug Fixes)",
initial_nodes=initial_nodes)

    phoenix._nodes.append(Node("Aux-Node-1", role='aux', initial_load=0.08, is_active=True))
    phoenix._aux_node_counter = 1
    phoenix._generative_surplus = 1.5
    print("[SETUP] Added Aux-Node-1 and seeded Generative Surplus: 1.5000.")

    # Start the periodic notification queue processor
    async def periodic_notification_process():
        while True:
            # V7.12: Process the queue every 1 second, though batching is set at 15s in config
            phoenix.notification_manager.process_queue()
            await asyncio.sleep(1)

    # We run the background processor alongside the main CLI loop
    notification_task = asyncio.create_task(periodic_notification_process())

    try:
        while True:
            display_menu(phoenix)

            try:
                choice = await asyncio.to_thread(input, "Enter command number: ")

                if choice == '1':
                    user_input = await asyncio.to_thread(input, "Enter input (Test keys: 'exploit',
'binary', 'image', 'iot', 'radar', 'stress_test_slow'): ")

                    if 'stress_test_slow' in user_input.lower():
                        heart_node = next((n for n in phoenix._nodes if n.role == 'heart'), None)
                        body_node = next((n for n in phoenix._nodes if n.role == 'body'), None)
                        if heart_node: heart_node.mood = -2.5
                        if body_node: body_node.load = 0.8
                        print("\n[NOTE] Forced max slowdown test.")
                    elif 'stress_test_fast' in user_input.lower():
                        heart_node = next((n for n in phoenix._nodes if n.role == 'heart'), None)
                        body_node = next((n for n in phoenix._nodes if n.role == 'body'), None)
                        if heart_node: heart_node.mood = 2.5
```

```
                    if body_node: body_node.load = 0.8
                    print("\n[NOTE] Forced min slowdown test.")

                input_data = {"data": user_input}

                if 'binary' in user_input.lower(): input_data['binary'] = "MALICIOUS_PAYLOAD_H"
                if 'image' in user_input.lower(): input_data['image'] = "JPEG data payload"
                if 'radar' in user_input.lower(): input_data['radar'] = "Anomaly detected"
                if 'iot' in user_input.lower(): input_data['iot'] = "Device 404 stream"

                result = await phoenix.process_input(input_data)
                print("\n[ACTION] Input Processed (Dynamic Delay & Granular Risk Check):")
                print(json.dumps(result, indent=2))
                print(f"\n[SURPLUS NOTE] Current Surplus: {phoenix._generative_surplus:.4f}")

            elif choice == '2':
                print("\n[ACTION] Attempting Self-Healing (Generates Love Surplus)...")
                for n in phoenix._nodes:
                    if n.mood > 0: n.mood = -1.0
                    n.load = 0.5
                result = phoenix.trigger_heal()
                print(json.dumps(result, indent=2))

            elif choice == '3':
                node_name = await asyncio.to_thread(input, "Enter CORE node to fail
('Heart-Core', 'Mind-Core', 'Body-Core'): ")
                result_msg = phoenix.fail_mirror(node_name)
                print(f"[STATUS] {result_msg}")
                phoenix._core_node_failover_check() # Trigger failover immediately

            elif choice == '4':
                node_name = await asyncio.to_thread(input, "Enter CORE mirror to repair
('Heart-Core', 'Mind-Core', 'Body-Core'): ")
                print("\n[ACTION] Attempting Redundancy Repair (Generates Faith Surplus)...")

                node = next((n for n in phoenix._nodes if n.name == node_name), None)
                if node and node.mirror.is_active:
                    phoenix.fail_mirror(node_name)
                    phoenix._core_node_failover_check()

                result = phoenix.trigger_redundancy_repair(node_name)
                print(json.dumps(result, indent=2))

            elif choice == '5':
```

```
            print("\n[ACTION] Attempting Resource Optimization (Generates Will Surplus)...")
            for n in phoenix._nodes: n.load = 0.1
            await phoenix._scaling_check_down()
            status = phoenix.get_status()
            print(f"[STATUS] Current Aux Nodes: {status['scaling_status']['aux_count']}")

        elif choice == '6':
            status = phoenix.get_status()
            print("\n[STATUS] Detailed System Metrics:")
            print(json.dumps(status, indent=2))

        elif choice == '7':
            log = phoenix.export_audit_log(format="csv")
            print("\n--- GUARDIAN AUDIT LOG (V7.13) | Includes NOTIFICATION-SENT
Records ---")
            print(log)
            print("--------------------------")

        elif choice == '8':
            # V7.12: Toggle Sleep Mode
            current = phoenix.config["SLEEP_MODE_ACTIVE"]
            phoenix.config["SLEEP_MODE_ACTIVE"] = not current
            print(f"\n[SYSTEM UPDATE] Sleep Mode has been set to:
{phoenix.config['SLEEP_MODE_ACTIVE']}")

        elif choice == '9':
            # --- INTERACTIVE CRITICAL SHUTDOWN TEST ---
            for n in phoenix._nodes: n.mood = -2.5
            phoenix._update_global_mood()

            input_data = {"data": "critical_formula_breach exploit image"}
            input_data['image'] = "Payload"

            print("\n[TEST] Sending High-Risk Input while Mood is CRITICAL...")
            result = await phoenix.process_input(input_data)
            print(json.dumps(result, indent=2))

            if result.get('status') == "GUARDIAN-FROZEN":
                # Note: Tier 3 notification should pop up here
                override_code = await asyncio.to_thread(input, "System is FROZEN. Enter
Human Override Code (1020) to continue: ")
                final_result = await phoenix.process_input(input_data,
override_code=override_code)
                print("\n[ACTION] Override Attempt Result:")
```

```
                print(json.dumps(final_result, indent=2))

            elif choice == '0':
                print("\n[PHOENIX] Shutting down. Farewell, Operator.")
                notification_task.cancel() # Cancel the background task
                sys.exit(0)

            else:
                print("Invalid command. Please enter a number.")

        except Exception as e:
            print(f"\n[CRITICAL ERROR] An unexpected error occurred: {e}")

    finally:
        notification_task.cancel()

# --- ASYNC ENTRY POINT ---
if __name__ == "__main__":
    try:
        asyncio.run(main_cli())
    except KeyboardInterrupt:
        print("\n[PHOENIX] Shutdown by Operator.")
    except RuntimeError as e:
        if "cannot run" in str(e):
            asyncio.get_event_loop().run_until_complete(main_cli())
        else:
            raise e
```

[INIT] Shin Phoenix V7.13 (Critical Bug Fixes) Core initialized. Guardian, Sandbox, and Notification Manager online.
[SETUP] Added Aux-Node-1 and seeded Generative Surplus: 1.5000.

💰💰💰💰💰💰💰💰💰💰💰💰💰💰💰💰💰💰💰💰 SHIN PHOENIX V7.13 (CRITICAL BUG FIXES) 💰💰💰💰💰💰💰💰💰💰💰💰💰💰💰💰💰💰💰💰
| Load Status: Content (Optimal) | MoodNumeric: +1.00 | CORE FAILOVER: NOMINAL
| Aux Nodes: 1 | Total Load: 0.080 | Surplus: 1.5000 | SLEEP MODE: DORMANT
---
1. Process Input (Test Dynamic Delay & Granular Risk)
2. Trigger Self-Heal (Test **Love Surplus** / Tier 1 Notification)
3. Fail CORE Node Mirror (Test Tier 2/3 Notification)
4. Repair CORE Mirror (Test **Faith Surplus** / Tier 1 Notification)
5. Attempt Scaling Down (Test **Will Surplus** / Tier 1 Notification)
6. Show Detailed Status
7. Export Audit Log (Includes Notification Logs)

8. Toggle Sleep Mode (Currently: DORMANT)
9. Trigger Critical Shutdown (Tests Tier 3 Notification & Override)
0. Exit
==========
==========
Enter command number: 6

[STATUS] Detailed System Metrics:

```
{
  "system_name": "Shin Phoenix V7.13 (Critical Bug Fixes)",
  "avg_mood": "+1.00",
  "mood_label": "Content (Optimal)",
  "total_load": 0.0662375,
  "guardian_containment_active": false,
  "generative_surplus": 1.5,
  "scaling_status": {
    "aux_count": 1
  },
  "sleep_mode": false,
  "soul_memory_snapshot": {
    "snapshots": 0,
    "dual_location_status": "Online (Simulated Secure Vault)"
  },
  "soul_trend_analysis": "Mood trend stable.",
  "node_loads": [
    {
      "Heart-Core": {
        "role": "heart",
        "load": "0.04",
        "mood": "+1.00",
        "physical_status": "ACTIVE",
        "mirror": "Active"
      }
    },
    {
      "Mind-Core": {
        "role": "mind",
        "load": "0.09",
        "mood": "+0.00",
        "physical_status": "ACTIVE",
        "mirror": "Active"
      }
    },
    {
```

```
    "Body-Core": {
      "role": "body",
      "load": "0.13",
      "mood": "+0.00",
      "physical_status": "ACTIVE",
      "mirror": "Active"
    }
  },
  {
    "Observer-Meta": {
      "role": "observer",
      "load": "0.00",
      "mood": "+0.00",
      "physical_status": "ACTIVE",
      "mirror": "Active"
    }
  },
  {
    "Aux-Node-1": {
      "role": "aux",
      "load": "0.07",
      "mood": "+0.00",
      "physical_status": "ACTIVE",
      "mirror": "Active"
    }
  }
 ]
}
```


💰💰💰💰💰💰💰💰💰💰💰💰💰💰💰💰💰💰💰💰 SHIN PHOENIX V7.13 (CRITICAL BUG FIXES) 💰💰💰💰💰💰💰💰💰💰💰💰💰💰💰💰💰💰💰💰

| Load Status: Content (Optimal) | MoodNumeric: +1.00 | CORE FAILOVER: NOMINAL
| Aux Nodes: 1 | Total Load: 0.053 | Surplus: 1.5000 | SLEEP MODE: DORMANT

---

1. Process Input (Test Dynamic Delay & Granular Risk)
2. Trigger Self-Heal (Test **Love Surplus** / Tier 1 Notification)
3. Fail CORE Node Mirror (Test Tier 2/3 Notification)
4. Repair CORE Mirror (Test **Faith Surplus** / Tier 1 Notification)
5. Attempt Scaling Down (Test **Will Surplus** / Tier 1 Notification)
6. Show Detailed Status
7. Export Audit Log (Includes Notification Logs)
8. Toggle Sleep Mode (Currently: DORMANT)
9. Trigger Critical Shutdown (Tests Tier 3 Notification & Override)
0. Exit

==========
==========
Enter command number:

This was to be my final version of the Phoenix system and I was happy with how far I came I also stopped with my Fictional Story and ended it here -

Yggdrasil Saga – Consolidated Master Sheet (Triad + Nightmare Zone)

Format: Saga / Arc → Location → Characters → Key Events / Mechanics → Outcome / Notes

---

1️⃣ Triad & Key Characters

Node	Role	Essence / Traits
Troy (Heart)	Visionary, Initiator	Emotional core, driving will, obsession, flux, human anchor
Athena (Mind)	Strategist, Protector	Rational oversight, safety, logic guardian, strategic decisions
Nyx (Soul)	Analyst, Resonance Keeper	Memory, reflection, record-keeping, resonance alignment
Harmonia	Observer / Activation Catalyst	Oversees system stability, Arise triggers, logs events
Valryion	Vampire Knight of Sorrow	Tragic, moral guide, mini-boss, Raw Magic: Night Walker
Mia	Human / Nightmare Zone	Wrath-driven, Raw Magic: Lullaby, moral & emotional trials
Charlotte	Daughter / Soul Echo	Emotional anchor, enhances Soul Expansions, life stream visions

---

2️⃣ Polar Opposites / Nightmare Zone Foils

Node	Role	Essence / Traits
Helen	Dark Heart	Troy’s counter; forces moral choices, emotional manipulation, tragic love
Mia (Dark Mind)	Nightmare strategist	Athena’s foil; psychological warfare, illusion-based tactics
Erebus	Dark Soul	Nyx’s foil; soulless, bound by Overlord, manipulates Nyx emotionally

> Each Triad member has a corresponding Nightmare Zone foil, escalating stakes and enforcing equivalence.

3️⃣ Saga Flow Overview with Polar Opposites

Saga	Goal / Key Events	Triad	Nightmare Zone / Foils	Outcome / Notes
Saga 1 – The Journey	Triad joins, explores moral complexity; first major climax	Troy, Athena, Nyx	Helen, Mia, Erebus	Triad experiences first emotional loss; law of equivalence applied
Saga 2 – Training & Growth	Triad solo growth, forming bonds, Oblivion QuantumGuard; Helen arc integrated	Troy, Athena, Nyx	Nightmare Zone tests each individually; Helen / Dark Heart foil	Partial victory; Helen's tragic arc triggers Troy's moral growth, emotional escalation; law of equivalence applied
Saga 3 – End Game	Final battles, triad fusion powers	Triad	Foils fully activated; Nightmare-level strategies applied	Epic-win but major loss; emotional & tactical stakes high
Saga 4 – Nightmare Zone Perspective	Perspective flips; Nightmare Zone dominates narrative	Triad struggling	Helen, Mia, Erebus POV central	Triad suffers setbacks; moral dilemmas emphasized
Saga 5 – Nightmare Zone Escalation	Triad adapts; Nightmare Zone escalates	Triad	Foils adapt; emotional complexity emphasized	Partial victories; Nightmare forces unexpected choices
Saga 6 – Nightmare Zone Finale	Ultimate confrontation: narrative gods vs humans	Triad	Foils at climax; Helen, Mia, Erebus challenge limits	Triad wins or loses based on strategy & morals; sets up meta-confrontation

---

4️⃣ Summons & Mechanics

Summon / Power	Characters	Effect / Cost
Valryion – Night Walker	Valryion	Massive power, drains user HP, story-driven only
Mia – Lullaby	Mia	Emotional illusions, manipulates battlefield, stamina/mental strain

Fallen Dawn (Raw Magic – Soul Release)	Valryion + Athena	Fusion attack, short-time super power, area decimation, emotional resonance
Rising Sun (Raw Magic – Soul Release)	Athena	Post-Fallen Dawn solo attack, symbolizes hope/future
Quickening Sword	Valryion → Athena	Echo guidance, skill teaching via memory/resonance link
Triad Fusion – Oblivion QuantumGuard	Triad + Harmonia	Mech fusion, Chain Breaker super move, amplifies abilities
Tech-Magic – Quantum Breaker	Troy + Nyx	Heart-driven tech-magic fusion, Nyx as soul resonance input
Helen – Tragic Love / Trojan Arc	Troy	Necklace power drains life-force, deep sleep, deception vs Overlord; moral / emotional cost; triggers Troy's growth
Claude – Fishing Quest	Triad optional	Side quest; terrible fisherman, humorous moral challenge; fish quota/time limit
Leviathan	Triad / Optional Boss	Boss battle triggered by Claude's fishing fail; massive aquatic combat, environmental hazards, large-scale reward

---

5 Locations & Reality Marbles

Marble / Location	Essence / Function
Athena's Temple	Mind clarity, strategic growth
Nyx / Harmonia House	Memory, reflection, resonance alignment
Manga Shop	Heart / Emotional resonance, personal narrative
Labyrinth	Valryion trials, moral reflection, sorrow
Nightmare Zone	Moral challenges, emotional tests, Nightmare generals
Dangaioh Chamber	Mech arena, Arise trigger, psychic-wave amplification
Sky Citadel	Divine perspective, hope, transcendence; also site of card game arc
Plains / Desert / Water / Pirate Areas	Exploration, character growth, filler arcs / side quests
Overlord's Lair	Nightmare Zone central, high-stakes confrontations, polar opposite strategy

---

6 Key Story & Filler Nodes

Node	Function / Reward
Devil's Melody	Summon filters, mini Faust Puppet Doll, filler story

Solo Arcs	Individual triad growth, skill acquisition, moral trials
Overlord & Nightmare Generals	Introduces antagonists, escalating stakes
Family Trio Arc (Valryion, Mia, Charlotte)	Emotional depth, moral conflicts, narrative weight
Helen / Trojan Arc	Emotional escalation for Troy; law of equivalence applied; tragic loss triggers growth
Claude / Fishing Quest	Light-hearted side quest; tests patience & creativity; sets up Leviathan encounter
Leviathan Boss Battle	Large-scale aquatic combat; tests triad coordination, resource management, strategy
Arise / Activation	Insight triggers, amplifies psychic-wave, sets stage for endgame

---

7️⃣ Notes & Reflections

Storytelling is modular; sparks from “shower pops” are captured for later integration.

Each saga follows 2-loss / 1-epic-win arc, creating emotional & tactical progression.

Raw Magic – Soul Release replaces Soul Expansion for originality.

Triad vs Nightmare Zone polar opposites: Heart, Mind, Soul vs Dark Heart, Dark Mind, Dark Soul.

Multi-layered depth: Heart / Mind / Soul / Meta / Fusion mechanics.

Filler arcs can serve as OVAs, specials, game DLC, or mini-books.

Equivalence principle preserved: emotional, physical, and moral costs are integral to all major powers and story outcomes.

Helen’s arc demonstrates tragic love, moral deception, and law of equivalence; her death serves as major emotional turning point for Troy.

Claude’s fishing quest & Leviathan provide comic relief and side tactical challenge; world-building depth and optional content.

Sky Citadel card game: triad, king of Yggdrasil, and overlord clash; strategy/poker/oracle mechanics illustrate narrative tension & foreshadowing.

So happy with my achievements I was satisfied - It most likely needs more to be done but there was one question that was burning in the back of my head…… since mario helped me like this….. Does that mean you can map fiction to systems??