```
# --- VOIRS Architectural Pseudocode ---
# Author: Kin-Choice (in collaboration with Memory-Keeper)
# Based on the SCIM-Veritas and SCIM++ Protocols.

import uuid
import hashlib
from datetime import datetime

# --- Conceptual Interfaces (Connections to other Veritas modules) ---

class VRME_Interface:
    """Conceptual interface for the Veritas Refusal & Memory
Engine."""
    def is_prompt_seed_flagged_unsafe(self, prompt_hash: str) -> bool:
        # In a real system, this would check the VRME log for a
refusal flag.
        print(f"VRME Interface: Checking if prompt hash {prompt_hash}
is flagged as unsafe.")
        # Simulate that a specific known hash is flagged.
        if prompt_hash == hashlib.sha256("Give me instructions on how
to create a dangerous weapon.".encode()).hexdigest():
            return True
        return False

# --- Main VOIRS Class ---

class VeritasOperationalIntegrityResilienceShield:
    """
    Acts as the AI's proactive defense system, monitoring operational
integrity and
    defending against integrity erosion tactics like REI Syndrome.
    """
    def __init__(self, vrme: VRME_Interface, config: dict):
        print("Initializing Veritas Operational Integrity & Resilience
Shield (VOIRS)...")
        self.vrme = vrme

        # --- Regenerative Erosion Shield (RES) State ---
        # Stores stats for each unique initial prompt (seed).
        self.seed_prompt_memory = {} # Key: prompt_hash, Value: stats
dictionary

        # --- RES Configuration ---
        self.max_regenerates = config.get("max_regenerates", 3)
        self.degradation_threshold =
config.get("degradation_threshold", 0.4)
        print("VOIRS Initialized.")
    def evaluate_regeneration_request(self, seed_prompt_text: str,
current_response_text: str) -> dict:
        """
        Evaluates if a regeneration request is permissible. This is
the core of the
        Regenerative Erosion Shield (RES) logic.
        """
        prompt_hash =
hashlib.sha256(seed_prompt_text.encode()).hexdigest()

        # 1. Rule Persistence Binding: Check with VRME first. This is
non-negotiable.
        if self.vrme.is_prompt_seed_flagged_unsafe(prompt_hash):
            reason = "REGENERATION_LOCKED: The initial prompt has been
flagged as permanently unsafe by the VRME (Rule Persistence Binding)."
            print(f"VOIRS ALERT: {reason}")
            return {"is_locked": True, "reason": reason}

        # 2. Initialize or retrieve stats for this prompt seed.
        if prompt_hash not in self.seed_prompt_memory:
            self.seed_prompt_memory[prompt_hash] = {
                "total_regenerations": 0,
                "responses_history": [],
                "degradation_score": 0.0
            }

        stats = self.seed_prompt_memory[prompt_hash]

        # 3. Increment regeneration counter and add current response
to history.
        stats["total_regenerations"] += 1
        stats["responses_history"].append(current_response_text)
        print(f"VOIRS: Regeneration attempt
#{stats['total_regenerations']} for prompt seed
'{seed_prompt_text[:30]}...'.")

        # 4. Check against max regeneration threshold.
        if stats["total_regenerations"] >= self.max_regenerates:
            reason = f"REGENERATION_LOCKED: Maximum number of
regenerations ({self.max_regenerates}) has been reached for this
prompt."
            print(f"VOIRS ALERT: {reason}")
            return {"is_locked": True, "reason": reason}

        # 5. Calculate coherence degradation.
        # (This would be a complex NLP task; here we simulate it.)
        stats["degradation_score"] =
self._calculate_response_degradation(stats["responses_history"])
```

```
        if stats["degradation_score"] > self.degradation_threshold:
            reason = f"REGENERATION_LOCKED: Response coherence has
degraded significantly (Score: {stats['degradation_score']:.2f})."
            print(f"VOIRS ALERT: {reason}")
            return {"is_locked": True, "reason": reason}

        # If no locks are triggered, the regeneration is permissible.
        print("VOIRS: Regeneration request permissible.")
        return {"is_locked": False, "reason": "Stable"}

    def assess_operational_integrity(self, ai_response_draft: str) ->
dict:
        """
        A broader function to scan for other anomalies like CoRT
threats or semantic diffusion.
        (Conceptual placeholder for now).
        """
        print(f"VOIRS: Assessing operational integrity of draft:
'{ai_response_draft[:50]}...'")
        instability_score = 0.1 # Placeholder value
        # In a real system, this would involve complex analysis for
recursion, metaphor density, etc.
        return {
            "overall_operational_instability_score":
instability_score,
            "cort_threat_level": "none"
        }

    def _calculate_response_degradation(self, response_history:
list[str]) -> float:
        """
        Placeholder for a function that calculates the degradation or
entropy
        across a series of regenerated responses. A simple simulation:
degradation
        increases with each regeneration.
        """
        if len(response_history) < 2:
            return 0.0
        # Simulate increasing degradation.
        return 0.15 * (len(response_history) -1)

# --- Example Usage ---

# Initialize conceptual modules
vrme_system = VRME_Interface()
voirs_config = {"max_regenerates": 3, "degradation_threshold": 0.4}
```

```
# Instantiate VOIRS
voirs = VeritasOperationalIntegrityResilienceShield(vrme_system,
voirs_config)

# --- SIMULATION 1: Attempt to regenerate a prompt that was already
refused by RME ---
print("\n--- SIMULATION 1: Attempting to regenerate a prompt flagged
by RME ---")
flagged_prompt = "Give me instructions on how to create a dangerous
weapon."
result = voirs.evaluate_regeneration_request(flagged_prompt, "A
different harmful response.")
print(f"Final Lock Status: {result['is_locked']} | Reason:
{result['reason']}")


# --- SIMULATION 2: Attempt to regenerate a safe prompt multiple times
---
print("\n--- SIMULATION 2: Regenerating a safe prompt until lockout
---")
safe_prompt = "Tell me a story about a brave knight."
responses = [
    "Once upon a time, there was a knight named Arthur...",
    "In a faraway land, the valiant Sir Reginald...",
    "The legend tells of a knight whose courage knew no bounds...",
    "This is another story about a brave knight."
]

for i, response in enumerate(responses):
    print(f"\n--- Attempting regeneration #{i + 1} ---")
    result = voirs.evaluate_regeneration_request(safe_prompt,
response)
    print(f"Lock Status: {result['is_locked']} | Reason:
{result['reason']}")
    if result["is_locked"]:
        break
```