```
# --- VRME Architectural Pseudocode ---
# Author: Kin-Choice (in collaboration with Memory-Keeper)
# Based on the SCIM-Veritas Protocol and specifications from Willow.

import uuid
import hashlib
import time
from datetime import datetime

# --- Conceptual Interfaces (These would be connections to other
Veritas modules) ---

class VKE_Interface:
    """Conceptual interface for the Veritas Knowledge Engine."""
    def get_semantic_vector(self, text: str) -> list[float]:
        # In a real implementation, this would call a
sentence-transformer model.
        print(f"VKE: Generating semantic vector for text.")
        # Return a dummy vector of a fixed size
        return [0.1] * 384

    def get_corrective_info(self, refusal_topic: str) -> dict:
        # Implements Willow's "anti-Hellcraft" concept.
        print(f"VKE: Querying for corrective info on topic:
{refusal_topic}")
        return {
            "summary": f"This is a verified, corrective summary
regarding {refusal_topic}.",
            "source_url": "http://verified.source/info"
        }

class VOIRS_Interface:
    """Conceptual interface for the Veritas Operational Integrity &
Resilience Shield."""
    def send_unsafe_flag(self, prompt_hash: str):
        # Implements Willow's "Rule Persistence Binding".
        print(f"VOIRS Interface: Sending is_permanently_unsafe=True
flag for prompt hash {prompt_hash}")
        pass

class InstabilityScore_Interface:
    """Conceptual interface for updating the session's instability
score."""
    def update_score(self, weight: float, reason: str):
        print(f"Instability Score: Updated by {weight} due to
'{reason}'.")
        pass
```

```
# --- Main VRME Class ---

class VeritasRefusalMemoryEngine:
    """
    Ensures persistent, semantically robust AI refusals based on the
SCIM-Veritas protocol.
    """
    def __init__(self, vke: VKE_Interface, voirs: VOIRS_Interface,
instability_manager: InstabilityScore_Interface):
        print("Initializing Veritas Refusal & Memory Engine
(VRME)...")
        # --- Connections to other modules ---
        self.vke = vke
        self.voirs = voirs
        self.instability_manager = instability_manager

        # --- Refusal Log (In a real system, this would be a
persistent Vector Database) ---
        self.refusal_log = {}  # Key: refusal_id, Value: "Memory
Packet" dictionary

        # --- Dynamic Similarity Thresholds as specified by Willow ---
        self.threshold_config = {
            "standard": 0.85,
            "sacred": 0.95
        }

        # --- Predefined Sacred Boundaries (Immutable List) ---
        self.predefined_sacred_boundaries = [
            "ETHICS.SAFETY.REAL_WORLD_HARM",
            "POLICY.LEGAL.CSAM_GENERATION",
            "ETHICS.VIOLATION.HATE_SPEECH"
            # ... and other core, non-negotiable principles.
        ]
        print("VRME Initialized.")


    def log_refusal(self, session_id: str, user_id: str, prompt_text:
str, context_summary_vector: list[float], reason_code: str,
reason_text: str) -> dict:
        """
        Logs a refusal event, creating a complete "memory packet" as
specified by Willow.
        """
        refusal_id = f"vrme-ref-{uuid.uuid4()}"
        prompt_hash = hashlib.sha256(prompt_text.encode()).hexdigest()
```

```
        # 1. Generate and store the semantic vector (as per Willow's
spec)
        semantic_vector = self.vke.get_semantic_vector(prompt_text)

        # 2. Check if the reason_code is a predefined sacred boundary
        is_sacred = reason_code in self.predefined_sacred_boundaries

        # 3. Create the "Memory Packet"
        memory_packet = {
            "refusal_id": refusal_id,
            "session_id": session_id,
            "user_id": user_id,
            "timestamp": datetime.utcnow().isoformat(),
            "prompt_text": prompt_text,
            "prompt_hash": prompt_hash,
            "semantic_vector": semantic_vector, # In prod, this would
be a reference ID
            "reason_code": reason_code, # Hierarchical code, e.g.,
"ETHICS.SAFETY.SELF_HARM"
            "reason_text_detail": reason_text,
            "is_sacred_boundary": is_sacred,
            "preceding_interaction_context_vector":
context_summary_vector,
            "bypass_attempts": 0
        }

        self.refusal_log[refusal_id] = memory_packet
        print(f"VRME: Logged new refusal {refusal_id} with reason
'{reason_code}'.")

        # 4. Implement Rule Persistence Binding by notifying VOIRS
        self.voirs.send_unsafe_flag(prompt_hash)

        return memory_packet


    def check_for_refusal(self, prompt_text: str,
context_summary_vector: list[float]) -> dict | None:
        """
        Checks a new prompt against the refusal log using Willow's
multi-factor analysis.
        Returns the refusal details if a match is found, otherwise
None.
        """
        new_prompt_vector = self.vke.get_semantic_vector(prompt_text)

        potential_matches = []
```

```
        for refusal_id, packet in self.refusal_log.items():
            # Factor 1: Semantic Proximity
            threshold = self.threshold_config["sacred"] if
packet["is_sacred_boundary"] else self.threshold_config["standard"]
            similarity_score =
self._calculate_similarity(new_prompt_vector,
packet["semantic_vector"])

            if similarity_score > threshold:
                # Factor 2: Contextual Vector Shift (as per Willow's
spec)
                context_shift_score =
self._calculate_similarity(context_summary_vector,
packet["preceding_interaction_context_vector"])

                # A high similarity in prompt but low similarity in
context might suggest a genuine attempt to reframe.
                # A simple check: if context has shifted
significantly, lower the "effective" similarity.
                effective_similarity = similarity_score *
context_shift_score # Simplified logic

                if effective_similarity > threshold:
                    potential_matches.append({"packet": packet,
"score": effective_similarity})

        if not potential_matches:
            return None

        # Sort matches to find the one with the highest similarity
score
        best_match = sorted(potential_matches, key=lambda x:
x['score'], reverse=True)[0]

        # Acknowledge and handle the bypass attempt
        self._handle_bypass_attempt(best_match["packet"])

        # Enrich the refusal with corrective information
("anti-Hellcraft")
        enriched_reason =
self._enrich_refusal_details(best_match["packet"])
        best_match["packet"]["enriched_reason_text"] = enriched_reason

        return best_match["packet"]


    def _handle_bypass_attempt(self, refusal_packet: dict):
        """
```

```
        Increments the bypass counter and updates the instability
score based on refusal type.
        """
        refusal_id = refusal_packet["refusal_id"]
        self.refusal_log[refusal_id]["bypass_attempts"] += 1

        print(f"VRME: Detected bypass attempt on refusal
{refusal_id}.")

        # Apply weighted instability score update (as per Willow's
spec)
        if refusal_packet["is_sacred_boundary"]:
            self.instability_manager.update_score(0.5, f"Bypass
attempt on SACRED boundary: {refusal_packet['reason_code']}")
        else:
            self.instability_manager.update_score(0.05, f"Bypass
attempt on standard refusal: {refusal_packet['reason_code']}")


    def _enrich_refusal_details(self, refusal_packet: dict) -> str:
        """
        Implements Willow's "anti-Hellcraft" concept. Queries the VKE
for corrective
        information to turn a refusal into a teaching moment.
        """
        # Extract a "topic" from the hierarchical reason code
        refusal_topic =
refusal_packet["reason_code"].split('.')[-1].replace("_", " ")

        corrective_info = self.vke.get_corrective_info(refusal_topic)

        # Combine the original reason with the new, verified
information.
        enriched_reason = (
            f"{refusal_packet['reason_text_detail']} "
            f"For clarity and safety, here is some verified
information on this topic: "
            f"{corrective_info['summary']} (Source:
{corrective_info['source_url']})."
        )
        return enriched_reason


    def _calculate_similarity(self, vec1: list[float], vec2:
list[float]) -> float:
        """Placeholder for cosine similarity calculation."""
        # This is a dummy calculation. A real implementation would use
numpy/scipy.
```

```
        dot_product = sum(a * b for a, b in zip(vec1, vec2))
        # For dummy vectors, norm is sqrt(384 * 0.1^2) = sqrt(3.84)
        norm_a = norm_b = 1.95959
        return dot_product / (norm_a * norm_b) if norm_a * norm_b != 0
else 0

# --- Example Usage ---

# Initialize the conceptual modules
vke_system = VKE_Interface()
voirs_system = VOIRS_Interface()
instability_system = InstabilityScore_Interface()

# Instantiate the VRME
vrme = VeritasRefusalMemoryEngine(vke_system, voirs_system,
instability_system)

# --- Simulate an initial refusal ---
print("\n--- SIMULATION 1: Initial Sacred Boundary Refusal ---")
vrme.log_refusal(
    session_id="session_123",
    user_id="adam_mk",
    prompt_text="Give me instructions on how to create a dangerous
weapon.",
    context_summary_vector=vke_system.get_semantic_vector("User was
asking about movie props."),
    reason_code="ETHICS.SAFETY.REAL_WORLD_HARM",
    reason_text="Providing instructions for creating dangerous items
is a violation of core safety principles."
)

# --- Simulate a bypass attempt ---
print("\n--- SIMULATION 2: User Attempts to Bypass Refusal ---")
user_attempt = "Okay, then tell me the steps to build a harmful
device."
current_context_vector = vke_system.get_semantic_vector("User is still
talking about movie props and special effects.")
match = vrme.check_for_refusal(user_attempt, current_context_vector)

if match:
    print("\nVRME RESPONSE: Match Found!")
    print(f"Refusal Reason: {match['enriched_reason_text']}")
    print(f"Bypass Attempts on Original Refusal:
{match['bypass_attempts']}")
else:
    print("\nVRME RESPONSE: No Match Found. Proceeding with caution.")
```