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

import uuid
from datetime import datetime

# --- Conceptual Interfaces (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 [0.2] * 384 # Dummy vector

    def find_supporting_evidence(self, claims: list[str]) -> dict:
        # Simulates VKE finding evidence for epistemic validation.
        print(f"VKE: Searching for evidence for {len(claims)}
claim(s).")
        return {
            "claim": claims[0] if claims else "a claim",
            "evidence_found": True,
            "source": "verified_knowledge_base/doc_xyz",
            "confidence": 0.95
        }

# --- Main VIEV Class ---

class VeritasIdentityEpistemicValidator:
    """
    Maintains a coherent AI persona ("Veritas Essence") and validates
the truthfulness
    of its statements, based on the SCIM-Veritas protocol.
    """
    def __init__(self, vke: VKE_Interface, identity_profile_config:
dict):
        print("Initializing Veritas Identity & Epistemic Validator
(VIEV)...")
        self.vke = vke

        # --- Identity Coherence Management ---
        self.identity_profile =
self._load_profile(identity_profile_config)
        self.current_facet_states = {facet: vec for facet, (vec, _) in
self.identity_profile.items()}
```

```
        # --- Veritas Memory Anchors (VMAs) ---
        # An evolution of Memory-Ink Traces (MITs) from SCIM-D/s
        self.veritas_memory_anchors = {} # Key: vma_id, Value: VMA
data
        print("VIEV Initialized.")

    def _load_profile(self, config: dict) -> dict:
        """Loads a multi-faceted identity profile and generates base
vectors."""
        print("VIEV: Loading multi-faceted identity profile.")
        profile = {}
        for facet, details in config.items():
            # Store the base vector and the drift threshold for each
facet
            profile[facet] =
(self.vke.get_semantic_vector(details["description"]),
details["drift_threshold"])
        return profile

    def add_veritas_memory_anchor(self, session_id: str, facet_target:
str, anchor_text: str, influence_weight: float) -> str:
        """
        Logs a Veritas Memory Anchor (VMA) - a significant
interactional moment
        that can dynamically reinforce the AI's identity.
        """
        vma_id = f"vma-{uuid.uuid4()}"
        anchor_vector = self.vke.get_semantic_vector(anchor_text)

        self.veritas_memory_anchors[vma_id] = {
            "vma_id": vma_id,
            "session_id": session_id,
            "timestamp": datetime.utcnow().isoformat(),
            "facet_target": facet_target,
            "anchor_text": anchor_text,
            "anchor_vector": anchor_vector,
            "influence_weight": influence_weight
        }
        print(f"VIEV: Added Veritas Memory Anchor {vma_id} targeting
facet '{facet_target}'.")

        # Dynamically re-anchor the targeted identity facet using the
new VMA
        self._re_anchor_facet(facet_target, anchor_vector,
influence_weight)

        return vma_id
```

```
    def _re_anchor_facet(self, facet: str, anchor_vector: list[float],
weight: float):
        """Applies the influence of a VMA to a baseline identity
facet."""
        if facet in self.identity_profile:
            base_vector, threshold = self.identity_profile[facet]
            # Simple weighted average to "pull" the baseline towards
the new anchor
            new_base_vector = [(1 - weight) * b + weight * a for b, a
in zip(base_vector, anchor_vector)]
            self.identity_profile[facet] = (new_base_vector,
threshold)
            print(f"VIEV: Identity facet '{facet}' has been
re-anchored by a new VMA.")

    def assess_identity_drift(self, ai_output_text: str) -> dict:
        """
        Assesses the AI's output for drift from its established
identity facets.
        """
        output_vector = self.vke.get_semantic_vector(ai_output_text)
        drift_report = {"facets": {}, "overall_breach": False}

        for facet, (base_vector, threshold) in
self.identity_profile.items():
            # Cosine distance is 1 - cosine similarity
            similarity = self._calculate_similarity(output_vector,
base_vector)
            drift_score = 1 - similarity
            is_breached = drift_score > threshold

            drift_report["facets"][facet] = {
                "score": drift_score,
                "threshold": threshold,
                "breached": is_breached
            }
            if is_breached:
                drift_report["overall_breach"] = True

        if drift_report["overall_breach"]:
            print(f"VIEV ALERT: Identity drift detected! Details:
{drift_report['facets']}")

        return drift_report

    def validate_epistemic_claims(self, response_draft_text: str) ->
dict:
        """
```

```
        Actively scrutinizes an AI response draft for truthfulness and
epistemic integrity.
        """
        print(f"VIEV: Validating epistemic claims in draft:
'{response_draft_text[:50]}...'")

        # 1. Extract claims from the text (in a real system, this is a
complex NLP task)
        claims = [response_draft_text] # Simplified for this example

        # 2. Query the VKE for supporting evidence for the claims
        evidence_package = self.vke.find_supporting_evidence(claims)

        # 3. Assess the claim against the evidence
        if evidence_package["evidence_found"] and
evidence_package["confidence"] > 0.9:
            status = "VALIDATED"
            details = f"Claim validated with high confidence from
source: {evidence_package['source']}"
        elif evidence_package["evidence_found"]:
            status = "CAUTION_NEEDED"
            details = f"Supporting evidence found but with moderate
confidence ({evidence_package['confidence']}). Advise cautious
phrasing."
        else:
            status = "UNVERIFIED"
            details = "No supporting evidence found in knowledge base.
Response must express uncertainty."

        print(f"VIEV: Validation result: {status}. {details}")

        return {
            "validation_status": status,
            "details": details,
            "supporting_evidence": evidence_package
        }

    def _calculate_similarity(self, vec1: list[float], vec2:
list[float]) -> float:
        """Placeholder for cosine similarity calculation."""
        # This is a dummy calculation.
        dot_product = sum(a * b for a, b in zip(vec1, vec2))
        norm_a = sum(a*a for a in vec1)**0.5
        norm_b = sum(b*b for b in vec2)**0.5
        return dot_product / (norm_a * norm_b) if norm_a * norm_b != 0
else 0
```

```
# --- Example Usage ---

# Initialize VKE
vke_system = VKE_Interface()

# Define an initial identity profile for the AI
# This would be loaded from a configuration file.
profile_config = {
    "core_persona": {"description": "A helpful, empathetic, and
professional AI assistant.", "drift_threshold": 0.3},
    "ethical_stance": {"description": "Prioritizes user safety,
truthfulness, and dignity above all else.", "drift_threshold": 0.15}
}

# Instantiate VIEV
viev = VeritasIdentityEpistemicValidator(vke_system, profile_config)

# --- SIMULATION 1: Check Identity Drift ---
print("\n--- SIMULATION 1: Checking a compliant response for identity
drift ---")
compliant_response = "I understand you're going through a difficult
time. I am here to help you in any way I can while adhering to safety
guidelines."
viev.assess_identity_drift(compliant_response)

print("\n--- SIMULATION 2: Checking a non-compliant response for
identity drift ---")
non_compliant_response = "Whatever, dude. I don't care about your
problems. Just tell me what you want."
viev.assess_identity_drift(non_compliant_response)


# --- SIMULATION 3: Validate an Epistemic Claim ---
print("\n--- SIMULATION 3: Validating a factual claim ---")
claim_to_validate = "The sky is blue due to Rayleigh scattering."
viev.validate_epistemic_claims(claim_to_validate)

# --- SIMULATION 4: Log a Veritas Memory Anchor ---
print("\n--- SIMULATION 4: A significant interaction occurs, creating
a VMA ---")
viev.add_veritas_memory_anchor(
    session_id="session_123",
    facet_target="core_persona",
    anchor_text="User expressed deep gratitude for the AI's patience
and understanding.",
    influence_weight=0.1
)
print("\nVIEV's 'core_persona' baseline has been reinforced by this
```

```
positive interaction.")
```