Assignment 3 Submission

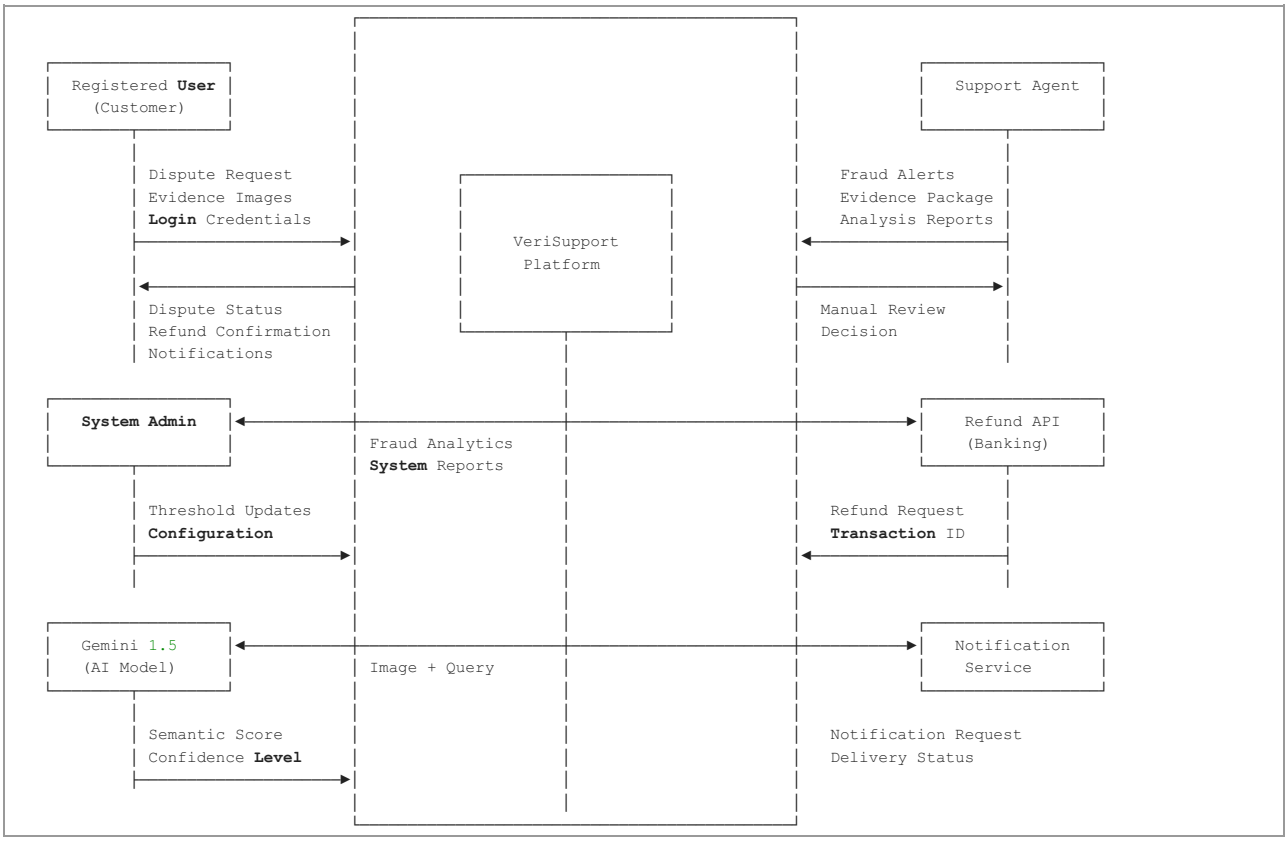
CS 331 Software Engineering Lab

VeriSupport: Autonomous Multimodal Customer Support & Forensic Integrity System

---

PART A: DATA FLOW DIAGRAMS

Q1. Context Diagram (Level 0 DFD)

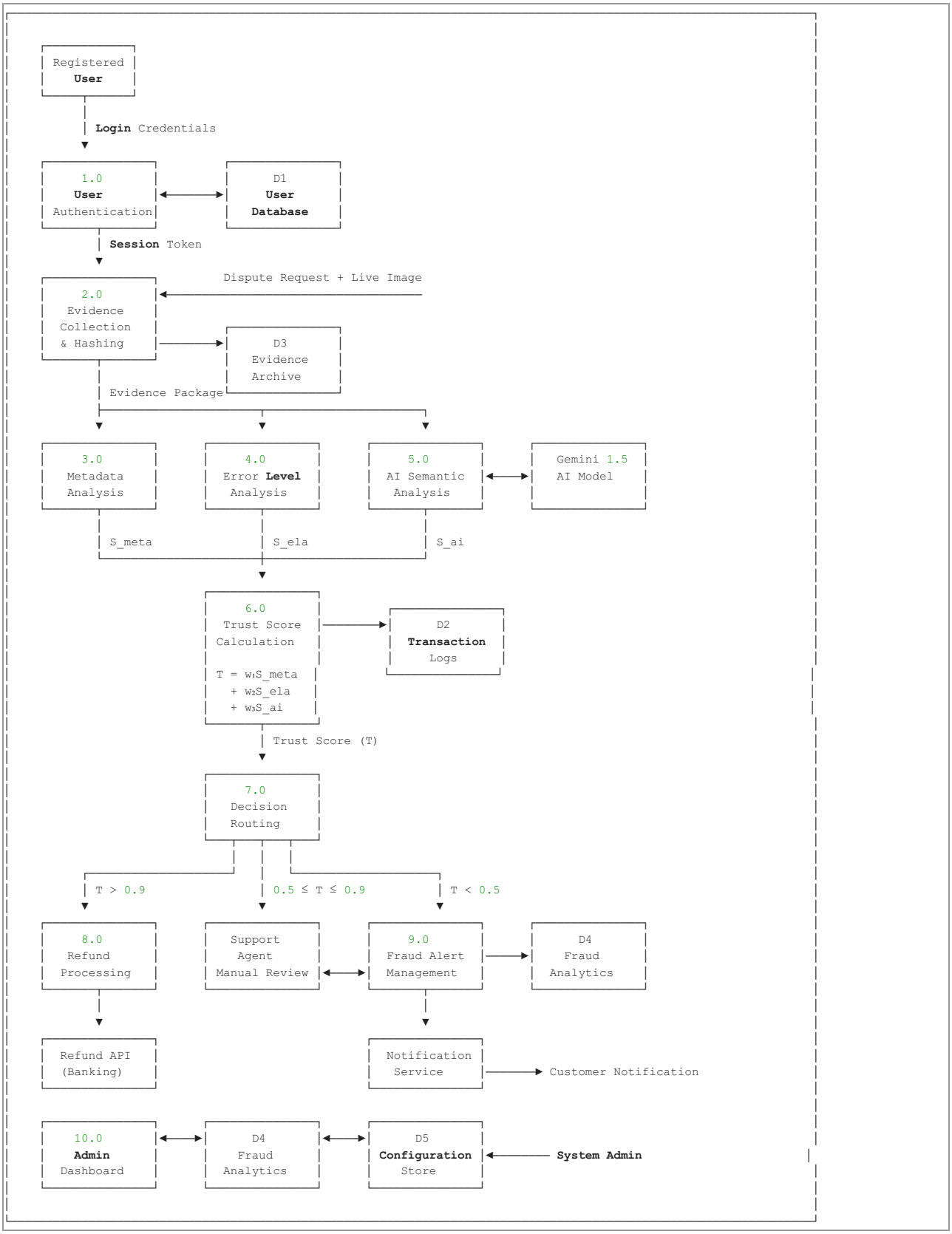


External Entities:

Entity	Type	Data Inflow	Data Outflow
Registered User	Primary Actor	Dispute request, Evidence images, Credentials	Status, Confirmation, Notifications
Support Agent	Secondary Actor	Manual review decisions	Fraud alerts, Evidence package, Reports
System Admin	Administrator	Threshold updates, Configuration	Fraud analytics, System reports
Refund API	External System	Refund status, Transaction ID	Refund request, Transaction details
Gemini 1.5 AI	External System	Semantic score, Confidence level	Image + context, Analysis request
Notification Service	External System	Delivery status	Notification requests

---

Q2. Level 1 DFD



Process Descriptions:

ID	Process	Description
1.0	User Authentication	Validates credentials, creates session tokens
2.0	Evidence Collection & Hashing	Captures live images, generates SHA-256 hash
3.0	Metadata Analysis	Parses EXIF data, detects software signatures
4.0	Error Level Analysis	Pixel-level compression artifact detection
5.0	AI Semantic Analysis	Uses Gemini 1.5 for semantic consistency
6.0	Trust Score Calculation	Computes weighted ensemble score
7.0	Decision Routing	Routes based on thresholds

8.0	Refund Processing	Interfaces with banking API
9.0	Fraud Alert Management	Creates alerts for agents
10.0	Admin Dashboard	Analytics and configuration

Data Stores:

ID	Name	Contents
D1	User Database	User profiles, credentials, sessions
D2	Transaction Logs	Dispute transactions, scores, decisions
D3	Evidence Archive	Hashed images, evidence metadata
D4	Fraud Analytics	Confirmed fraud cases, patterns
D5	Configuration Store	Thresholds, weights, parameters

PART B: UML CLASS DIAGRAM

Q1. Key Classes with Attributes and Methods

1. User (Abstract Base Class)

2. RegisteredUser

3. SupportAgent

4. SystemAdmin

5. RefundDispute

6. Evidence

7. ForensicEngine

8. MetadataAnalyzer

9. ELAProcessor

10. AISemanticChecker

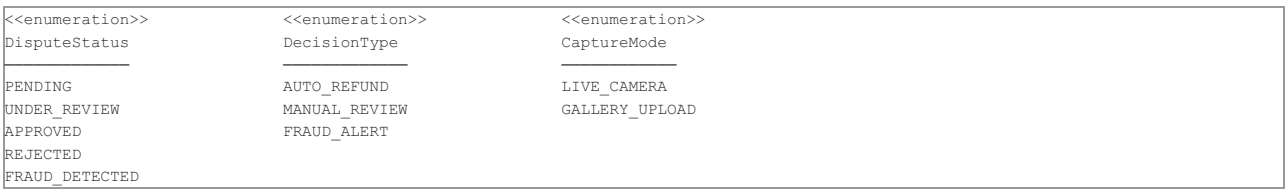
11. TrustScoreCalculator

12. DecisionRouter

13. RefundAPI (Interface)

14. NotificationService (Interface)

Enumerations:

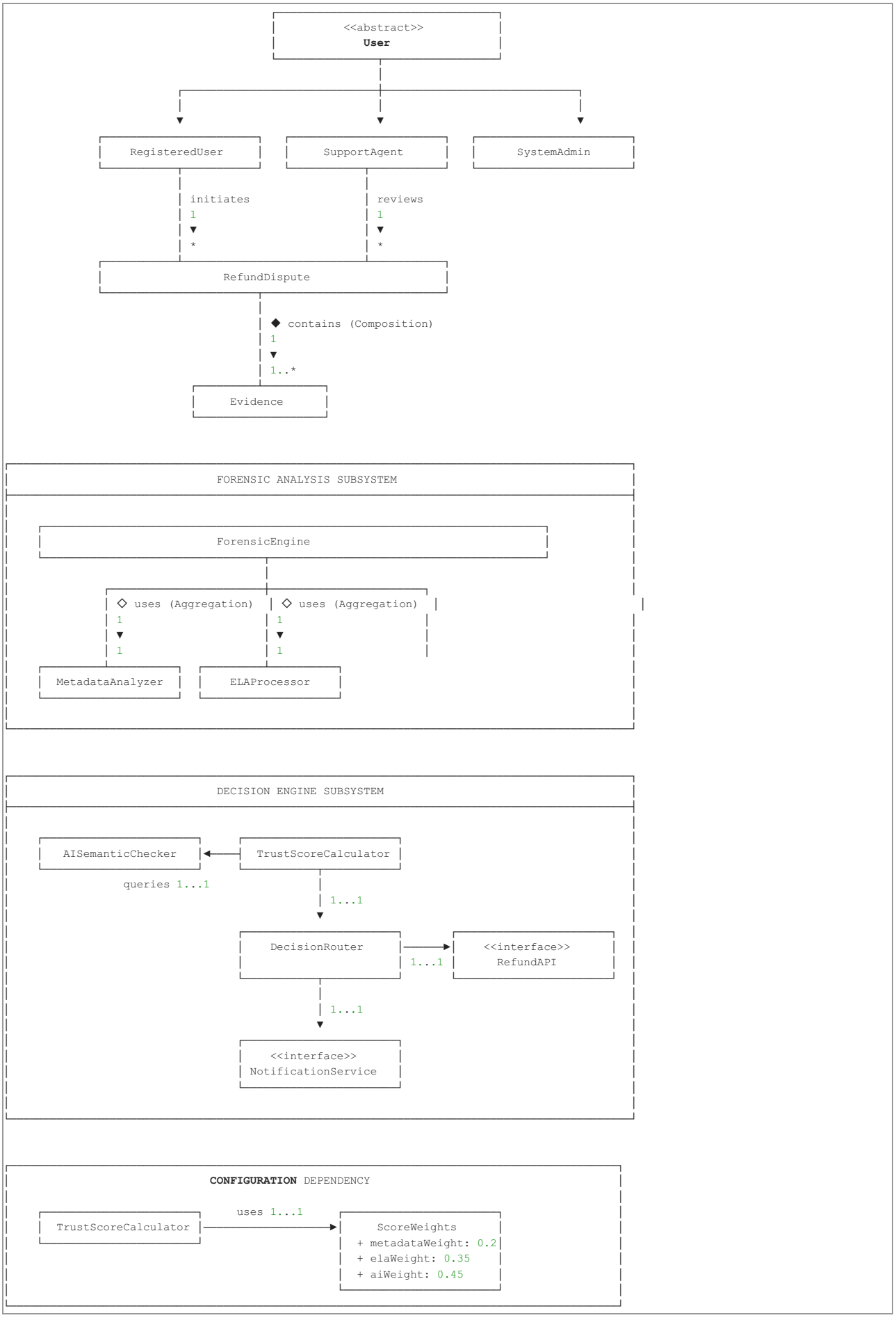


Visibility Legend:

Symbol	Visibility	Access Level
+	public	Accessible from anywhere
-	private	Accessible only within class

#	protected Accessible within class and subclasses

Q2. UML Class Diagram with Relationships



Relationship Summary Table:

Relationship Type	From Class	To Class	Cardinality	Description
Inheritance	RegisteredUser	User	-	User subclass for customers
Inheritance	SupportAgent	User	-	User subclass for agents
Inheritance	SystemAdmin	User	-	User subclass for admins
Composition	RefundDispute	Evidence	1 : 1..*	Dispute owns its evidence (lifecycle bound)
Aggregation	ForensicEngine	MetadataAnalyzer	1 : 1	Engine uses analyzer (independent lifecycle)
Aggregation	ForensicEngine	ELAProcessor	1 : 1	Engine uses processor (independent lifecycle)
Association	RegisteredUser	RefundDispute	1 : 0..*	User initiates disputes
Association	SupportAgent	RefundDispute	1 : 0..*	Agent reviews disputes
Association	TrustScoreCalculator	AISemanticChecker	1 : 1	Calculator queries AI
Association	DecisionRouter	RefundAPI	1 : 1	Router uses API
Association	DecisionRouter	NotificationService	1 : 1	Router sends notifications
Dependency	TrustScoreCalculator	ScoreWeights	1 : 1	Uses weight configuration

---

PART C: IMPLEMENTATION OF KEY MODULES

Module 1: Forensic Analysis Engine

File: forensic_engine.py

```
"""
VeriSupport - Forensic Analysis Engine Module

This module implements the Forensic Analysis Engine for detecting manipulated images.
- MetadataAnalyzer: Parses EXIF data and detects software signatures
- ELAProcessor: Performs Error Level Analysis for compression artifact detection
- ForensicEngine: Orchestrates both analyses
"""

from dataclasses import dataclass
from typing import Optional, Dict, List, Tuple
from enum import Enum
import hashlib
import io

# Optional imports for image processing
try:
    from PIL import Image, ImageChops
    import numpy as np
    HAS_IMAGING = True
except ImportError:
    HAS_IMAGING = False


class AnalysisStatus(Enum):
    """Status of forensic analysis."""
    PENDING = "pending"
    COMPLETED = "completed"
    FAILED = "failed"


@dataclass
class ExifData:
    """Container for extracted EXIF metadata."""
    make: Optional[str] = None
    model: Optional[str] = None
    software: Optional[str] = None
    datetime_original: Optional[str] = None
    raw_data: Optional[Dict] = None


@dataclass
class ELAResult:
    """Result of Error Level Analysis."""
    variance_score: float  # 0.0 - 1.0, lower is more authentic
    max_difference: float
    mean_difference: float
    suspicious_regions: List[Tuple[int, int, int, int]]


@dataclass
class ForensicResult:
    """Combined result of all forensic analyses."""
    metadata_score: float  # 0.0 or 1.0 (binary)
    ela_score: float  # 0.0 - 1.0, normalized
    image_hash: str
    status: AnalysisStatus
    flags: List[str]
    details: Dict
```

```python
class MetadataAnalyzer:
    """
    Analyzes image metadata (EXIF) to detect potential manipulation.
    Checks for software signatures (Adobe, GIMP, Stable Diffusion, etc.)
    """

    SUSPICIOUS_SOFTWARE: List[str] = [
        "adobe", "photoshop", "gimp", "canva", "pixlr",
        "stable diffusion", "midjourney", "dall-e", "dalle",
        "figma", "sketch", "affinity", "paint.net", "lightroom"
    ]

    def __init__(self):
        self._analysis_count = 0

    def extract_exif(self, image_data: bytes) -> ExifData:
        """Extract EXIF metadata from image bytes."""
        if not HAS_IMAGING:
            return ExifData(raw_data={"error": "Imaging library not available"})

        try:
            img = Image.open(io.BytesIO(image_data))
            exif_dict = {}
            if hasattr(img, '_getexif') and img._getexif():
                exif_dict = img._getexif()

            return ExifData(
                make=exif_dict.get(271),
                model=exif_dict.get(272),
                software=exif_dict.get(305),
                datetime_original=exif_dict.get(36867),
                raw_data=exif_dict if exif_dict else None
            )
        except Exception as e:
            return ExifData(raw_data={"error": str(e)})

    def detect_software_signatures(self, exif: ExifData) -> Tuple[bool, List[str]]:
        """Detect suspicious software signatures in metadata."""
        detected = []
        if exif.software:
            software_lower = exif.software.lower()
            for suspicious in self.SUSPICIOUS_SOFTWARE:
                if suspicious in software_lower:
                    detected.append(exif.software)
                    break
        return (len(detected) > 0, detected)

    def validate_source(self, exif: ExifData) -> float:
        """Validate if image appears from authentic source. Returns 0.0-1.0."""
        score = 0.5
        if exif.make and exif.model:
            score += 0.3
        if exif.datetime_original:
            score += 0.2
        is_suspicious, _ = self.detect_software_signatures(exif)
        if is_suspicious:
            score = 0.0
        if not exif.raw_data or exif.raw_data.get("error"):
            score = max(0.0, score - 0.3)
        return min(1.0, max(0.0, score))

    def analyze(self, image_data: bytes) -> Tuple[float, Dict]:
        """Perform complete metadata analysis."""
        self._analysis_count += 1
        exif = self.extract_exif(image_data)
        is_suspicious, detected_software = self.detect_software_signatures(exif)
        source_score = self.validate_source(exif)
        metadata_score = 0.0 if is_suspicious else (1.0 if source_score > 0.5 else 0.0)

        details = {
            "make": exif.make, "model": exif.model, "software": exif.software,
            "is_suspicious": is_suspicious, "detected_software": detected_software,
            "has_metadata": exif.raw_data is not None
        }
        return (metadata_score, details)


class ELAProcessor:
    """
    Performs Error Level Analysis (ELA) to detect image manipulation.

    Algorithm:
    1. Resave image at known quality (90%)
    2. Calculate pixel differences: Difference = |Pixel_A - Pixel_B|
    3. Amplify differences (x50)
```

```
    4. Authentic images = uniform noise; manipulated regions = bright glow
    """

    def __init__(self, resave_quality: int = 90, amplification_factor: int = 50):
        self._resave_quality = resave_quality
        self._amplification_factor = amplification_factor

    @property
    def resave_quality(self) -> int:
        return self._resave_quality

    @property
    def amplification_factor(self) -> int:
        return self._amplification_factor

    def perform_ela(self, image_data: bytes) -> ELAResult:
        """Perform Error Level Analysis on an image."""
        if not HAS_IMAGING:
            return ELAResult(variance_score=0.5, max_difference=0,
                             mean_difference=0, suspicious_regions=[])

        try:
            original = Image.open(io.BytesIO(image_data))
            if original.mode != 'RGB':
                original = original.convert('RGB')

            # Resave at known quality
            buffer = io.BytesIO()
            original.save(buffer, format='JPEG', quality=self._resave_quality)
            buffer.seek(0)
            resaved = Image.open(buffer)

            # Calculate and amplify differences
            diff = ImageChops.difference(original, resaved)
            diff_array = np.array(diff, dtype=np.float32)
            amplified = np.clip(diff_array * self._amplification_factor, 0, 255)

            # Compute variance score
            variance = np.var(amplified)
            variance_score = 1.0 - min(1.0, variance / 5000)

            return ELAResult(
                variance_score=variance_score,
                max_difference=float(np.max(amplified)),
                mean_difference=float(np.mean(amplified)),
                suspicious_regions=[]
            )
        except Exception:
            return ELAResult(variance_score=0.5, max_difference=0,
                             mean_difference=0, suspicious_regions=[])


class ForensicEngine:
    """Orchestrates forensic analysis by combining metadata and ELA analysis."""

    def __init__(self, ela_quality: int = 90, ela_amplification: int = 50):
        self._metadata_analyzer = MetadataAnalyzer()
        self._ela_processor = ELAProcessor(ela_quality, ela_amplification)

    @property
    def metadata_analyzer(self) -> MetadataAnalyzer:
        return self._metadata_analyzer

    @property
    def ela_processor(self) -> ELAProcessor:
        return self._ela_processor

    def analyze_evidence(self, image_data: bytes) -> ForensicResult:
        """Perform complete forensic analysis on submitted evidence."""
        flags = []
        image_hash = hashlib.sha256(image_data).hexdigest()

        # Metadata analysis
        try:
            metadata_score, metadata_details = self._metadata_analyzer.analyze(image_data)
            if metadata_details.get("is_suspicious"):
                flags.append("SUSPICIOUS_SOFTWARE_DETECTED")
            if not metadata_details.get("has_metadata"):
                flags.append("METADATA_STRIPPED")
        except Exception as e:
            metadata_score = 0.5
            metadata_details = {"error": str(e)}
            flags.append("METADATA_ANALYSIS_FAILED")

        # ELA analysis
        try:
```

```
            ela_result = self._ela_processor.perform_ela(image_data)
            ela_score = ela_result.variance_score
        except Exception:
            ela_score = 0.5
            flags.append("ELA_ANALYSIS_FAILED")

        status = AnalysisStatus.FAILED if "FAILED" in " ".join(flags) else AnalysisStatus.COMPLETED

        return ForensicResult(
            metadata_score=metadata_score, ela_score=ela_score,
            image_hash=image_hash, status=status, flags=flags,
            details={"metadata": metadata_details}
        )

    def run_full_scan(self, image_data: bytes) -> Tuple[float, float]:
        """Quick scan returning metadata and ELA scores."""
        result = self.analyze_evidence(image_data)
        return (result.metadata_score, result.ela_score)
```

Module 2: Trust Score Calculator

File: trust_score_calculator.py

```
"""
VeriSupport - Trust Score Calculator Module

Implements the decision-making engine:
- ScoreWeights: Configuration for weighted ensemble
- TrustScoreCalculator: Computes T = w1(S_meta) + w2(S_ela) + w3(S_ai)
- DecisionRouter: Routes decisions based on score thresholds
"""

from dataclasses import dataclass
from typing import Optional, Dict, List
from enum import Enum
from datetime import datetime


class DecisionType(Enum):
    """Types of decisions the system can make."""
    AUTO_REFUND = "auto_refund"
    MANUAL_REVIEW = "manual_review"
    FRAUD_ALERT = "fraud_alert"


@dataclass
class ScoreWeights:
    """Configuration for weighted trust score calculation."""
    metadata_weight: float = 0.20
    ela_weight: float = 0.35
    ai_weight: float = 0.45

    def validate(self) -> bool:
        """Validate weights sum to 1.0 and are non-negative."""
        total = self.metadata_weight + self.ela_weight + self.ai_weight
        all_positive = all(w >= 0 for w in [self.metadata_weight,
                                            self.ela_weight, self.ai_weight])
        return abs(total - 1.0) < 0.001 and all_positive


@dataclass
class TrustScoreResult:
    """Result of trust score calculation."""
    trust_score: float
    decision: DecisionType
    component_scores: Dict[str, float]
    weights_used: ScoreWeights
    calculation_timestamp: datetime
    confidence_level: str


class TrustScoreCalculator:
    """
    Calculates trust scores using weighted ensemble algorithm.

    Formula: T = w1(S_meta) + w2(S_ela) + w3(S_ai)

    Decision Thresholds:
    - T > 0.9: Auto-Refund
    - T < 0.5: Fraud Alert
    - 0.5 <= T <= 0.9: Manual Review
    """

    DEFAULT_AUTO_REFUND_THRESHOLD = 0.9
    DEFAULT_FRAUD_ALERT_THRESHOLD = 0.5
```

```python
    def __init__(self, weights: Optional[ScoreWeights] = None,
                 auto_refund_threshold: float = DEFAULT_AUTO_REFUND_THRESHOLD,
                 fraud_alert_threshold: float = DEFAULT_FRAUD_ALERT_THRESHOLD):
        self._weights = weights or ScoreWeights()
        self._auto_refund_threshold = auto_refund_threshold
        self._fraud_alert_threshold = fraud_alert_threshold

        if not self._weights.validate():
            raise ValueError("Invalid weights: must be non-negative and sum to 1.0")

    @property
    def weights(self) -> ScoreWeights:
        return self._weights

    @property
    def auto_refund_threshold(self) -> float:
        return self._auto_refund_threshold

    @property
    def fraud_alert_threshold(self) -> float:
        return self._fraud_alert_threshold

    def set_weights(self, w1: float, w2: float, w3: float) -> None:
        """Update the score weights."""
        new_weights = ScoreWeights(w1, w2, w3)
        if not new_weights.validate():
            raise ValueError(f"Weights must sum to 1.0, got {w1 + w2 + w3}")
        self._weights = new_weights

    def calculate_trust_score(self, metadata_score: float,
                              ela_score: float, ai_score: float) -> TrustScoreResult:
        """
        Calculate final trust score using weighted ensemble.

        T = w1(S_meta) + w2(S_ela) + w3(S_ai)
        """
        # Normalize scores to [0, 1]
        s_meta = max(0.0, min(1.0, metadata_score))
        s_ela = max(0.0, min(1.0, ela_score))
        s_ai = max(0.0, min(1.0, ai_score))

        # Calculate weighted trust score
        trust_score = (
            self._weights.metadata_weight * s_meta +
            self._weights.ela_weight * s_ela +
            self._weights.ai_weight * s_ai
        )
        trust_score = max(0.0, min(1.0, trust_score))

        # Determine decision
        decision = self.get_recommended_action(trust_score)

        # Calculate confidence
        scores = [s_meta, s_ela, s_ai]
        mean = sum(scores) / 3
        variance = sum((s - mean) ** 2 for s in scores) / 3
        confidence = "high" if variance < 0.05 else ("medium" if variance < 0.15 else "low")

        return TrustScoreResult(
            trust_score=trust_score, decision=decision,
            component_scores={"metadata": s_meta, "ela": s_ela, "ai": s_ai},
            weights_used=self._weights, calculation_timestamp=datetime.now(),
            confidence_level=confidence
        )

    def get_recommended_action(self, score: float) -> DecisionType:
        """Get recommended action based on trust score."""
        if score > self._auto_refund_threshold:
            return DecisionType.AUTO_REFUND
        elif score < self._fraud_alert_threshold:
            return DecisionType.FRAUD_ALERT
        return DecisionType.MANUAL_REVIEW


@dataclass
class FraudAlert:
    """Fraud alert for support agent review."""
    alert_id: str
    dispute_id: str
    trust_score: float
    flags: List[str]
    priority: str
    created_at: datetime
```

```python
class DecisionRouter:
    """
    Routes decisions based on trust scores.

    Routes:
    - Score > 0.9: Auto-refund via RefundAPI
    - Score < 0.5: Fraud alert to support agent
    - 0.5 <= Score <= 0.9: Manual review queue
    """

    def __init__(self):
        self._routing_log: List[Dict] = []

    def route_decision(self, dispute_id: str, trust_result: TrustScoreResult,
                       user_email: Optional[str] = None,
                       order_amount: float = 0.0) -> Dict:
        """Route decision based on trust score result."""
        decision = trust_result.decision

        result = {
            "dispute_id": dispute_id,
            "decision": decision.value,
            "trust_score": trust_result.trust_score,
            "timestamp": datetime.now().isoformat(),
            "action_taken": None
        }

        if decision == DecisionType.AUTO_REFUND:
            result["action_taken"] = "auto_refund_processed"
            print(f"[AutoRefund] Processing for {dispute_id}: ${order_amount:.2f}")

        elif decision == DecisionType.FRAUD_ALERT:
            alert = self.create_fraud_alert(dispute_id, trust_result)
            result["action_taken"] = "fraud_alert_created"
            result["alert_id"] = alert.alert_id

        else:
            result["action_taken"] = "queued_for_manual_review"
            print(f"[ManualReview] Queuing {dispute_id} (score: {trust_result.trust_score:.2f})")

        self._routing_log.append(result)
        return result

    def create_fraud_alert(self, dispute_id: str,
                           trust_result: TrustScoreResult) -> FraudAlert:
        """Create fraud alert for support agent review."""
        if trust_result.trust_score < 0.2:
            priority = "high"
        elif trust_result.trust_score < 0.35:
            priority = "medium"
        else:
            priority = "low"

        alert = FraudAlert(
            alert_id=f"ALERT-{dispute_id[:8]}-{datetime.now().strftime('%H%M%S')}",
            dispute_id=dispute_id,
            trust_score=trust_result.trust_score,
            flags=self._generate_flags(trust_result),
            priority=priority,
            created_at=datetime.now()
        )
        print(f"[FraudAlert] Created {alert.alert_id} (priority: {priority})")
        return alert

    def _generate_flags(self, trust_result: TrustScoreResult) -> List[str]:
        """Generate flags based on component scores."""
        flags = []
        scores = trust_result.component_scores
        if scores.get("metadata", 1) < 0.3:
            flags.append("SUSPICIOUS_METADATA")
        if scores.get("ela", 1) < 0.4:
            flags.append("IMAGE_MANIPULATION_DETECTED")
        if scores.get("ai", 1) < 0.5:
            flags.append("SEMANTIC_INCONSISTENCY")
        return flags


# Demo execution
if __name__ == "__main__":
    print("=" * 50)
    print("VeriSupport Trust Score Calculator - Demo")
    print("=" * 50)

    calculator = TrustScoreCalculator()
    router = DecisionRouter()
```

```python
# Test scenarios
scenarios = [
    ("Authentic", 1.0, 0.95, 0.92),   # High scores = auto-refund
    ("Fraudulent", 0.0, 0.3, 0.4),    # Low scores = fraud alert
    ("Borderline", 0.5, 0.7, 0.6),    # Medium = manual review
]

for name, meta, ela, ai in scenarios:
    result = calculator.calculate_trust_score(meta, ela, ai)
    print(f"\n{name}: Score={result.trust_score:.3f} -> {result.decision.value}")
    router.route_decision(f"DISP-{name.upper()}", result, order_amount=49.99)
```
