```
"""
Layer 1: Anonymous Submission Gateway
Handles pseudonym generation, encryption, hash chains, and metadata stripping.

Input: Raw evidence file + metadata
Output: Anonymized submission with pseudonym and encrypted evidence
"""

import hashlib
import logging
from datetime import datetime
from pathlib import Path
from typing import Dict, Optional, Tuple
from uuid import uuid4

from PIL import Image
from backend.services.metrics_service import MetricsService

# Initialize logger
logger = logging.getLogger(__name__)


class Layer1Anonymity:
    """
    Layer 1: Anonymous Submission Gateway

    Implements:
    - SHA-256 pseudonym generation
    - AES-256 file encryption
    - Hash chain entry creation
    - EXIF metadata stripping
    - Submission data anonymization
    """

    def __init__(
        self,
        crypto_service,
        hash_chain_service,
        metadata_service,
        storage_service,
        metrics_service: Optional[MetricsService] = None
    ):
        """
        Initialize Layer 1 with required services.

        Args:
```

```
            crypto_service: Cryptography service for encryption
            hash_chain_service: Hash chain management service
            metadata_service: Metadata stripping service
            storage_service: Storage service for file operations
        """
        self.crypto = crypto_service
        self.hash_chain = hash_chain_service
        self.metadata = metadata_service
        self.storage = storage_service
        self.metrics = metrics_service

        logger.info("Layer 1 (Anonymity) initialized")

    def process(
        self,
        submission_id: str,
        file_path: Path,
        evidence_type: str,
        text_narrative: Optional[str] = None,
        metadata: Optional[Dict] = None
    ) -> Dict:
        """
        Process submission through anonymity layer.

        Args:
            submission_id: Unique submission identifier
            file_path: Path to evidence file
            evidence_type: Type of evidence (image/audio/video)
            text_narrative: Optional text narrative
            metadata: Optional metadata dictionary

        Returns:
            dict: Anonymized submission data

        Raises:
            ValueError: If processing fails
        """
        logger.info(f"Layer 1 processing submission {submission_id}")

        try:
            # Step 1: Generate pseudonym
            pseudonym = self._generate_pseudonym(submission_id)
            logger.debug(f"Generated pseudonym: {pseudonym}")

            # Step 2: Strip metadata from evidence file
            cleaned_file_path = self._strip_metadata(
```

```
    file_path,
    evidence_type
)
logger.debug(f"Metadata stripped from evidence")

if self.metrics:
    self.metrics.record_anonymity_violation()

# Step 3: Compute evidence hash (before encryption)
evidence_hash = self._compute_evidence_hash(cleaned_file_path)
logger.debug(f"Evidence hash: {evidence_hash[:16]}...")

# Step 4: Encrypt evidence file
encrypted_file_path = self._encrypt_evidence(
    cleaned_file_path,
    submission_id
)
logger.debug(f"Evidence encrypted")

# Step 5: Add entry to hash chain
# Use correct method add_block which generates its own timestamp
block = self.hash_chain.add_block(
    submission_id=submission_id,
    evidence_hash=evidence_hash
)
chain_hash = block['hash']
logger.debug(f"Chain hash: {chain_hash[:16]}...")

# Step 6: Sanitize text narrative
sanitized_narrative = self._sanitize_text(text_narrative)

# Step 7: Anonymize metadata
anonymized_metadata = self._anonymize_metadata(metadata)

# Step 8: Build anonymized submission data
anonymized_data = {
    "id": submission_id,
    "pseudonym": pseudonym,
    "evidence_hash": evidence_hash,
    "chain_hash": chain_hash,
    "evidence_type": evidence_type,
    "text_narrative": sanitized_narrative,
    "metadata": anonymized_metadata,
    "encrypted_file_path": str(encrypted_file_path),
    "original_file_path": str(file_path),
    "timestamp_anonymized": datetime.utcnow().isoformat(),
```

```
            "anonymity_version": "1.0",
            "layer1_status": "completed"
        }

        logger.info(
            f"Layer 1 completed for {submission_id} "
            f"(pseudonym: {pseudonym})"
        )

        return anonymized_data

    except Exception as e:
        logger.error(f"Layer 1 processing failed for {submission_id}: {e}", exc_info=True)

        if self.metrics:
            self.metrics.record_anonymity_violation()

        raise ValueError(f"Anonymity processing failed: {str(e)}")

def _generate_pseudonym(self, submission_id: str) -> str:
    """
    Generate anonymous pseudonym from submission ID.

    Uses SHA-256 hash truncated to 16 characters for anonymity.

    Args:
        submission_id: Unique submission identifier

    Returns:
        str: 16-character pseudonym
    """
    # Generate salt from submission ID
    salt = submission_id.encode('utf-8')

    # Use crypto service to generate pseudonym
    pseudonym = self.crypto.generate_pseudonym(salt)

    return pseudonym

def _strip_metadata(
    self,
    file_path: Path,
    evidence_type: str
) -> Path:
    """
    Strip EXIF and other metadata from evidence file.
```

```
    Args:
        file_path: Path to original file
        evidence_type: Type of evidence

    Returns:
        Path: Path to cleaned file
    """
    try:
        if evidence_type == "image":
            return self.metadata.strip_image_metadata(file_path)
        elif evidence_type == "audio":
            return self.metadata.strip_audio_metadata(file_path)
        elif evidence_type == "video":
            return self.metadata.strip_video_metadata(file_path)
        else:
            # For text or unknown types, return original
            logger.warning(f"Unknown evidence type: {evidence_type}")
            return file_path

    except Exception as e:
        logger.warning(
            f"Metadata stripping failed, using original: {e}"
        )
        # Fail gracefully - use original file
        return file_path

def _compute_evidence_hash(self, file_path: Path) -> str:
    """
    Compute SHA-256 hash of evidence file.

    Args:
        file_path: Path to evidence file

    Returns:
        str: SHA-256 hash (hex)
    """
    return self.crypto.hash_file(file_path)

def _encrypt_evidence(
    self,
    file_path: Path,
    submission_id: str
) -> Path:
    """
    Encrypt evidence file using AES-256.
```

```
    Args:
        file_path: Path to file to encrypt
        submission_id: Submission identifier

    Returns:
        Path: Path to encrypted file
    """
    # Read file content
    with open(file_path, 'rb') as f:
        content = f.read()

    # Encrypt content
    encrypted_content = self.crypto.encrypt_data(content)

    # Save encrypted file
    encrypted_path = file_path.parent / f"{submission_id}_encrypted.bin"
    with open(encrypted_path, 'wb') as f:
        f.write(encrypted_content)

    return encrypted_path

def _sanitize_text(self, text: Optional[str]) -> Optional[str]:
    """
    Sanitize text narrative to remove identifying information.

    Args:
        text: Raw text narrative

    Returns:
        str: Sanitized text
    """
    if not text:
        return None

    # Remove HTML tags
    import re
    text = re.sub(r'<[^>]+>', '', text)

    # Remove control characters
    text = re.sub(r'[\x00-\x1f\x7f-\x9f]', '', text)

    # Remove excessive whitespace
    text = re.sub(r'\s+', ' ', text).strip()

    # Optional: Redact common PII patterns (phone, email, etc.)
```

```
        # Phone numbers
        text = re.sub(
            r'\b\d{3}[-.]?\d{3}[-.]?\d{4}\b',
            '[PHONE_REDACTED]',
            text
        )

        # Email addresses
        text = re.sub(
            r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b',
            '[EMAIL_REDACTED]',
            text
        )

        # Limit length
        max_length = 5000
        if len(text) > max_length:
            text = text[:max_length] + "... [TRUNCATED]"

        return text

    def _anonymize_metadata(self, metadata: Optional[Dict]) -> Dict:
        """
        Anonymize metadata by removing identifying fields.

        Args:
            metadata: Raw metadata dictionary

        Returns:
            dict: Anonymized metadata
        """
        if not metadata:
            return {}

        # Allowed metadata fields (whitelist approach)
        allowed_fields = {
            'incident_date',
            'location_city',
            'location_country',
            'category',
            'severity',
            'evidence_count'
        }

        # Filter metadata to only allowed fields
        anonymized = {
```

```
            k: v for k, v in metadata.items()
            if k in allowed_fields
        }

        # Redact precise locations (keep only city/country)
        if 'location_address' in metadata:
            anonymized['location_note'] = 'Address provided but redacted'

        return anonymized

    def decrypt_evidence(
        self,
        encrypted_file_path: Path,
        output_path: Optional[Path] = None
    ) -> Path:
        """
        Decrypt evidence file for processing.

        Args:
            encrypted_file_path: Path to encrypted file
            output_path: Optional output path for decrypted file

        Returns:
            Path: Path to decrypted file
        """
        # Read encrypted content
        with open(encrypted_file_path, 'rb') as f:
            encrypted_content = f.read()

        # Decrypt content
        decrypted_content = self.crypto.decrypt_data(encrypted_content)

        # Determine output path
        if not output_path:
            output_path = encrypted_file_path.parent / \
                          f"{encrypted_file_path.stem}_decrypted"

        # Save decrypted file
        with open(output_path, 'wb') as f:
            f.write(decrypted_content)

        logger.debug(f"Evidence decrypted to {output_path}")

        return output_path

    def verify_hash_chain(self, submission_id: str, evidence_hash: str) -> bool:
```

```
        """
        Verify submission exists in hash chain with correct hash.

        Args:
            submission_id: Submission identifier
            evidence_hash: Expected evidence hash

        Returns:
            bool: True if verification succeeds
        """
        try:
            return self.hash_chain.verify_entry(submission_id, evidence_hash)
        except Exception as e:
            logger.error(f"Hash chain verification failed: {e}")
            return False

    def get_chain_proof(self, submission_id: str) -> Dict:
        """
        Get hash chain proof for submission.

        Args:
            submission_id: Submission identifier

        Returns:
            dict: Chain proof data
        """
        try:
            return self.hash_chain.get_proof(submission_id)
        except Exception as e:
            logger.error(f"Failed to get chain proof: {e}")
            return {}

"""
Layer 2: Credibility Assessment Engine
Deepfake detection using pre-trained models with test-time augmentation.

Input: Anonymized submission
Output: Credibility scores and confidence intervals
"""

import logging
import time
from pathlib import Path
from tkinter import Image
from typing import Dict, List, Optional, Tuple
```

```
import numpy as np
import torch
from scipy import stats

from backend.services.metrics_service import MetricsService

# Initialize logger
logger = logging.getLogger(__name__)


class Layer2Credibility:
    """
    Layer 2: Credibility Assessment Engine

    Implements:
    - CLIP-based deepfake detection
    - Wav2Vec2 audio analysis
    - BLIP image captioning for consistency
    - Cross-modal consistency checks
    - Test-time augmentation (10 variations)
    - Confidence interval calculation
    """

    def __init__(
        self,
        storage_service,
        validation_service,
        image_utils,
        audio_utils,
        metrics_service: Optional[MetricsService] = None,
        device: Optional[str] = None
    ):
        """
        Initialize Layer 2 with parameters matching Orchestrator.
        Models are lazy-loaded on first use.

        Args:
            storage_service: Storage service
            validation_service: Validation service
            image_utils: Image utils
            audio_utils: Audio utils
            metrics_service: Metrics service
            device: Device for inference
        """
        self.storage = storage_service
        self.validation = validation_service
```

```
        self.image_utils = image_utils
        self.audio_utils = audio_utils
        self.metrics = metrics_service

        # Determine device
        if device:
            self.device = device
        else:
            self.device = "cuda" if torch.cuda.is_available() else "cpu"

        # Lazy loaded models
        self._clip = None
        self._wav2vec = None
        self._blip = None
        self._sentence_transformer = None

        logger.info(f"Layer 2 (Credibility) initialized on {self.device} (Lazy Loading)")

    @property
    def clip(self):
        if self._clip is None:
            logger.info("Loading CLIP model...")
            # Mock loading for MVP or import actual loader
            # For MVP we'll need real implementation from backend.models if available
            # But based on context, we should use what was passed before or mock it
            # Assuming models.py handles loading
            try:
                from backend.models import load_clip_model
                self._clip = load_clip_model(self.device)
            except ImportError:
                # Fallback or mock
                logger.warning("Could not load CLIP model, using mock")
                self._clip = self._create_mock_model("CLIP")
        return self._clip

    @property
    def wav2vec(self):
        if self._wav2vec is None:
            logger.info("Loading Wav2Vec2 model...")
            try:
                from backend.models import load_wav2vec_model
                self._wav2vec = load_wav2vec_model(self.device)
            except ImportError:
                logger.warning("Could not load Wav2Vec2 model, using mock")
                self._wav2vec = self._create_mock_model("Wav2Vec2")
        return self._wav2vec
```

```
    @property
    def blip(self):
        if self._blip is None:
            logger.info("Loading BLIP model...")
            try:
                from backend.models import load_blip_model
                self._blip = load_blip_model(self.device)
            except ImportError:
                logger.warning("Could not load BLIP model, using mock")
                self._blip = self._create_mock_model("BLIP")
        return self._blip

    @property
    def sentence_transformer(self):
        if self._sentence_transformer is None:
            logger.info("Loading Sentence Transformer...")
            try:
                from backend.models import load_sentence_transformer
                self._sentence_transformer = load_sentence_transformer(self.device)
            except ImportError:
                logger.warning("Could not load Sentence Transformer, using mock")
                self._sentence_transformer =
self._create_mock_model("SentenceTransformer")
        return self._sentence_transformer

    def _create_mock_model(self, name):
        """Create mock model for testing/fallback."""
        class MockModel:
            def predict_authenticity(self, image): return 0.8
            def extract_features(self, audio): return np.random.rand(1024)
            def generate_caption(self, image): return "A photo of corruption"
            def encode(self, text): return np.random.rand(384)
        return MockModel()

    def process(
        self,
        submission_id: str,
        file_path: Path,
        evidence_type: str,
        text_narrative: Optional[str] = None,
        use_augmentation: bool = True
    ) -> Dict:
        """
        Process submission through credibility assessment.
```

```
    Args:
        submission_id: Unique submission identifier
        file_path: Path to evidence file
        evidence_type: Type of evidence (image/audio/video)
        text_narrative: Optional text narrative
        use_augmentation: Whether to use test-time augmentation

    Returns:
        dict: Credibility assessment results

    Raises:
        ValueError: If assessment fails
    """
    logger.info(f"Layer 2 processing submission {submission_id}")
    start_time = time.time()

    try:
        # Route to appropriate assessment method
        if evidence_type == "image":
            scores = self._assess_image(
                file_path,
                text_narrative,
                use_augmentation
            )
        elif evidence_type == "audio":
            scores = self._assess_audio(
                file_path,
                text_narrative,
                use_augmentation
            )
        elif evidence_type == "video":
            scores = self._assess_video(
                file_path,
                text_narrative,
                use_augmentation
            )
        elif evidence_type == "text":
            scores = self._assess_text(text_narrative)
        else:
            raise ValueError(f"Unsupported evidence type: {evidence_type}")

        # Calculate confidence intervals
        confidence_interval = self._calculate_confidence_interval(
            scores['deepfake_scores']
        )
```

```
        # Aggregate final score
        final_score = self._aggregate_scores(scores)

        if self.metrics:
            self.metrics.update_credibility_score(final_score)

            # Check if deepfake detected (score < 0.5 typically means fake)
            if final_score < 0.5:
                self.metrics.record_deepfake_detection()

        # Build result
        result = {
            "submission_id": submission_id,
            "deepfake_score": final_score,
            "deepfake_scores_raw": scores['deepfake_scores'],
            "consistency_score": scores.get('consistency_score', 1.0),
            "plausibility_score": scores.get('plausibility_score', 1.0),
            "final_score": final_score,
            "confidence_interval": confidence_interval,
            "entropy": self._calculate_entropy(scores['deepfake_scores']),
            "augmentation_count": len(scores['deepfake_scores']),
            "processing_time": time.time() - start_time,
            "layer2_status": "completed",
            "timestamp_assessed": time.time()
        }

        # Flag for human review if high uncertainty
        if result['entropy'] > 0.4:
            result['requires_human_review'] = True
            logger.warning(
                f"High uncertainty (entropy={result['entropy']:.3f}) "
                f"for {submission_id}"
            )
        else:
            result['requires_human_review'] = False

        logger.info(
            f"Layer 2 completed for {submission_id} "
            f"(score={final_score:.3f}, took {result['processing_time']:.2f}s)"
        )

        return result

    except Exception as e:
        logger.error(
            f"Layer 2 processing failed for {submission_id}: {e}",
```

```
            exc_info=True
        )
        raise ValueError(f"Credibility assessment failed: {str(e)}")

def _assess_image(
    self,
    image_path: Path,
    text_narrative: Optional[str],
    use_augmentation: bool
) -> Dict:
    """
    Assess image credibility using CLIP and BLIP.

    Args:
        image_path: Path to image file
        text_narrative: Optional narrative
        use_augmentation: Use test-time augmentation

    Returns:
        dict: Assessment scores
    """
    deepfake_scores = []

    # Load image
    from PIL import Image
    image = Image.open(image_path).convert('RGB')

    # Test-time augmentation
    if use_augmentation:
        augmented_images = self._augment_image(image)
    else:
        augmented_images = [image]

    # Run CLIP inference on each augmentation
    for aug_image in augmented_images:
        try:
            score = self.clip.predict_authenticity(aug_image)
            deepfake_scores.append(score)
        except Exception as e:
            logger.warning(f"CLIP inference failed: {e}")

    # If no scores obtained, use neutral score
    if not deepfake_scores:
        logger.warning("No deepfake scores obtained, using neutral 0.5")
        deepfake_scores = [0.5]
```

```
        # Cross-modal consistency check
        consistency_score = 1.0
        if text_narrative:
            try:
                # Generate caption from image
                caption = self.blip.generate_caption(image)

                # Compare caption with narrative
                consistency_score = self._compute_text_similarity(
                    caption,
                    text_narrative
                )
                logger.debug(
                    f"Caption: '{caption}' | "
                    f"Consistency: {consistency_score:.3f}"
                )
            except Exception as e:
                logger.warning(f"Consistency check failed: {e}")

        import time
        inference_start = time.time()

        # Physical plausibility check
        plausibility_score = self._check_image_plausibility(image)

        inference_time = time.time() - inference_start

        if self.metrics:
            self.metrics.record_model_inference("clip", inference_time)

        return {
            'deepfake_scores': deepfake_scores,
            'consistency_score': consistency_score,
            'plausibility_score': plausibility_score
        }

    def _assess_audio(
        self,
        audio_path: Path,
        text_narrative: Optional[str],
        use_augmentation: bool
    ) -> Dict:
        """
        Assess audio credibility using Wav2Vec2.

        Args:
```

```
    audio_path: Path to audio file
    text_narrative: Optional narrative
    use_augmentation: Use test-time augmentation

Returns:
    dict: Assessment scores
"""
deepfake_scores = []

# Extract audio features
try:
    features = self.wav2vec.extract_features(audio_path)

    # Simple heuristic: check feature statistics
    # Real audio typically has more variance
    feature_variance = np.var(features)

    # Normalize to [0, 1] range
    # Higher variance !' more likely real
    score = min(1.0, feature_variance / 0.1)

    deepfake_scores.append(score)

    # If augmentation enabled, add noise and re-check
    if use_augmentation:
        for _ in range(4):
            noisy_score = score + np.random.normal(0, 0.05)
            noisy_score = np.clip(noisy_score, 0, 1)
            deepfake_scores.append(noisy_score)

except Exception as e:
    logger.warning(f"Audio assessment failed: {e}")
    deepfake_scores = [0.5]

import time
inference_start = time.time()

# Consistency with narrative
consistency_score = 0.8 if text_narrative else 1.0

# Plausibility (simple check for audio length)
plausibility_score = 1.0

inference_time = time.time() - inference_start

if self.metrics:
```

```
            self.metrics.record_model_inference("clip", inference_time)


        return {
            'deepfake_scores': deepfake_scores,
            'consistency_score': consistency_score,
            'plausibility_score': plausibility_score
        }

    def _assess_video(
        self,
        video_path: Path,
        text_narrative: Optional[str],
        use_augmentation: bool
    ) -> Dict:
        """
        Assess video credibility (simplified - sample frames).

        Args:
            video_path: Path to video file
            text_narrative: Optional narrative
            use_augmentation: Use test-time augmentation

        Returns:
            dict: Assessment scores
        """
        # MVP: Sample middle frame and assess as image
        # Full implementation would analyze multiple frames

        try:
            # Extract middle frame
            from backend.utils.image_utils import extract_video_frame
            middle_frame_path = extract_video_frame(
                video_path,
                frame_number="middle"
            )

            import time
            inference_start = time.time()
            score = self._assess_image(
                middle_frame_path,
                text_narrative,
                use_augmentation
            )
            inference_time = time.time() - inference_start
```

```
            if self.metrics:
                self.metrics.record_model_inference("clip", inference_time)

            # Assess as image
            return score

        except Exception as e:
            logger.error(f"Video assessment failed: {e}")
            return {
                'deepfake_scores': [0.5],
                'consistency_score': 1.0,
                'plausibility_score': 1.0
            }

    def _assess_text(self, text_narrative: str) -> Dict:
        """
        Assess text-only submission.

        Args:
            text_narrative: Text narrative

        Returns:
            dict: Assessment scores
        """
        # For text-only, assign neutral credibility
        # Can be enhanced with linguistic analysis

        deepfake_scores = [0.7]  # Slightly positive default

        # Check text length and coherence
        if len(text_narrative) < 50:
            deepfake_scores[0] -= 0.2

        return {
            'deepfake_scores': deepfake_scores,
            'consistency_score': 1.0,
            'plausibility_score': 1.0
        }

    def _augment_image(self, image: Image) -> List[Image]:
        """
        Apply test-time augmentation to image.

        Args:
            image: PIL Image
```

```
Returns:
    list: List of augmented images (including original)
"""
from torchvision import transforms

augmentations = [
    # Original
    transforms.Compose([]),

    # Brightness
    transforms.ColorJitter(brightness=0.2),
    transforms.ColorJitter(brightness=0.3),

    # Contrast
    transforms.ColorJitter(contrast=0.2),

    # Saturation
    transforms.ColorJitter(saturation=0.2),

    # Gaussian blur
    transforms.GaussianBlur(kernel_size=3),

    # Rotation
    transforms.RandomRotation(degrees=5),
    transforms.RandomRotation(degrees=10),

    # Horizontal flip
    transforms.RandomHorizontalFlip(p=1.0),

    # Combined
    transforms.Compose([
        transforms.ColorJitter(brightness=0.1, contrast=0.1),
        transforms.RandomRotation(degrees=3)
    ])
]

augmented_images = []
for aug in augmentations:
    try:
        aug_image = aug(image)
        augmented_images.append(aug_image)
    except Exception as e:
        logger.warning(f"Augmentation failed: {e}")

# Ensure at least original image
if not augmented_images:
```

```
            augmented_images = [image]

        return augmented_images

    def _compute_text_similarity(self, text1: str, text2: str) -> float:
        """
        Compute semantic similarity between two texts.

        Args:
            text1: First text
            text2: Second text

        Returns:
            float: Similarity score [0, 1]
        """
        try:
            # Get embeddings
            emb1 = self.sentence_transformer.encode(text1)
            emb2 = self.sentence_transformer.encode(text2)

            # Compute cosine similarity
            from sklearn.metrics.pairwise import cosine_similarity
            similarity = cosine_similarity(
                emb1.reshape(1, -1),
                emb2.reshape(1, -1)
            )[0][0]

            # Normalize to [0, 1]
            similarity = (similarity + 1) / 2

            return float(similarity)

        except Exception as e:
            logger.warning(f"Text similarity computation failed: {e}")
            return 0.5  # Neutral default

    def _check_image_plausibility(self, image: Image) -> float:
        """
        Check physical plausibility of image.

        Simple heuristics for MVP:
        - Check resolution
        - Check color distribution
        - Check edge sharpness

        Args:
```

```
            image: PIL Image

        Returns:
            float: Plausibility score [0, 1]
        """
        score = 1.0

        try:
            # Check resolution (too low is suspicious)
            width, height = image.size
            if width < 100 or height < 100:
                score -= 0.3

            # Check if image is too small for detail
            total_pixels = width * height
            if total_pixels < 50000:  # < ~224x224
                score -= 0.2

            # Check color distribution
            img_array = np.array(image)
            if len(img_array.shape) == 3:
                # Check if colors are too uniform
                color_std = np.std(img_array)
                if color_std < 10:  # Very uniform
                    score -= 0.2

            # Ensure score in [0, 1]
            score = max(0.0, min(1.0, score))

        except Exception as e:
            logger.warning(f"Plausibility check failed: {e}")
            score = 1.0  # Fail open

        return score

    def _aggregate_scores(self, scores: Dict) -> float:
        """
        Aggregate multiple scores into final credibility score.

        Args:
            scores: Dictionary of score arrays

        Returns:
            float: Aggregated score [0, 1]
        """
        deepfake_mean = np.mean(scores['deepfake_scores'])
```

```
        consistency = scores.get('consistency_score', 1.0)
        plausibility = scores.get('plausibility_score', 1.0)

        # Weighted average
        # Deepfake detection: 60%
        # Consistency: 25%
        # Plausibility: 15%
        final_score = (
            0.60 * deepfake_mean +
            0.25 * consistency +
            0.15 * plausibility
        )

        return float(np.clip(final_score, 0, 1))

    def _calculate_confidence_interval(
        self,
        scores: List[float],
        confidence: float = 0.90
    ) -> Tuple[float, float]:
        """
        Calculate confidence interval for scores.

        Args:
            scores: List of scores
            confidence: Confidence level (default 0.90 for 90% CI)

        Returns:
            tuple: (lower_bound, upper_bound)
        """
        if len(scores) < 2:
            # Not enough data for CI
            mean = scores[0] if scores else 0.5
            return (mean, mean)

        mean = np.mean(scores)
        std_err = stats.sem(scores)

        # Calculate CI using t-distribution
        ci = stats.t.interval(
            confidence,
            len(scores) - 1,
            loc=mean,
            scale=std_err
        )
```

```
        # Clip to [0, 1]
        lower = max(0.0, ci[0])
        upper = min(1.0, ci[1])

        return (float(lower), float(upper))

    def _calculate_entropy(self, scores: List[float]) -> float:
        """
        Calculate entropy of score distribution.

        High entropy indicates high uncertainty.

        Args:
            scores: List of scores

        Returns:
            float: Normalized entropy [0, 1]
        """
        if len(scores) < 2:
            return 0.0

        # Bin scores into histogram
        hist, _ = np.histogram(scores, bins=10, range=(0, 1), density=True)

        # Normalize histogram
        hist = hist / np.sum(hist)

        # Calculate entropy
        entropy = -np.sum(hist * np.log(hist + 1e-10))

        # Normalize to [0, 1] (max entropy for uniform is log(10))
        max_entropy = np.log(10)
        normalized_entropy = entropy / max_entropy

        return float(normalized_entropy)

"""
Layer 3: Coordination Detection System
Detects coordinated attacks using graph analysis and stylometric features.

Input: Submission with credibility scores
Output: Coordination flags and community detection results
"""

import logging
from collections import Counter
```

```
from datetime import datetime, timedelta
from pathlib import Path
from typing import Dict, List, Optional, Set, Tuple

import networkx as nx
import numpy as np
from networkx.algorithms import community
from sklearn.svm import OneClassSVM
from sklearn.preprocessing import StandardScaler

from backend.services.metrics_service import MetricsService


# Initialize logger
logger = logging.getLogger(__name__)


class Layer3Coordination:
    """
    Layer 3: Coordination Detection System

    Implements:
    - Graph construction with multiple edge types (style, temporal, content)
    - Stylometric feature extraction (TTR, lexical density, POS patterns)
    - Louvain community detection
    - One-Class SVM for anomaly detection
    - Coordination flagging with confidence scores
    """

    def __init__(
        self,
        storage_service,
        text_utils,
        graph_utils,
        min_similarity: float = 0.7,
        time_window_hours: int = 24,
        min_community_size: int = 3,
        metrics_service: Optional[MetricsService] = None
    ):
        """
        Initialize Layer 3 with configuration.

        Args:
            storage_service: Storage service for accessing submissions
            text_utils: Text utilities for stylometric analysis
            graph_utils: Graph utilities for NetworkX operations
```

```
            min_similarity: Minimum similarity threshold for edges
            time_window_hours: Time window for temporal edges
            min_community_size: Minimum size for flagged communities
        """
        self.storage = storage_service
        self.text_utils = text_utils
        self.graph_utils = graph_utils
        self.metrics = metrics_service
        self.min_similarity = min_similarity
        self.time_window = timedelta(hours=time_window_hours)
        self.min_community_size = min_community_size

        # Initialize One-Class SVM for anomaly detection
        self.anomaly_detector = None
        self.scaler = StandardScaler()

        logger.info(
            f"Layer 3 (Coordination) initialized "
            f"(similarity>={min_similarity}, window={time_window_hours}h)"
        )

    def process(
        self,
        submission_id: str,
        text_narrative: Optional[str] = None,
        timestamp: Optional[datetime] = None
    ) -> Dict:
        """
        Process submission through coordination detection.

        Args:
            submission_id: Unique submission identifier
            text_narrative: Text narrative for stylometric analysis
            timestamp: Submission timestamp

        Returns:
            dict: Coordination detection results

        Raises:
            ValueError: If detection fails
        """
        logger.info(f"Layer 3 processing submission {submission_id}")

        try:
            # Step 1: Get all recent submissions
            recent_submissions = self._get_recent_submissions(timestamp)
```

```
logger.debug(f"Found {len(recent_submissions)} recent submissions")

# Step 2: Extract stylometric features for current submission
current_features = self._extract_stylometric_features(
    submission_id,
    text_narrative
)

# Step 3: Build coordination graph
graph = self._build_coordination_graph(
    recent_submissions,
    submission_id,
    current_features,
    text_narrative
)

# Step 4: Detect communities
communities = self._detect_communities(graph)
logger.debug(f"Detected {len(communities)} communities")

# Step 5: Check if current submission is in suspicious community
coordination_info = self._check_coordination(
    submission_id,
    communities,
    graph,
    current_features,
    recent_submissions
)

# Step 6: Anomaly detection (if enough data)
anomaly_score = self._detect_anomaly(
    current_features,
    recent_submissions
)

result = {
    "submission_id": submission_id,
    "flagged": coordination_info['flagged'],
    "confidence": coordination_info['confidence'],
    "community_id": coordination_info.get('community_id'),
    "community_size": coordination_info.get('community_size'),
    "similarity_scores": coordination_info.get('similarity_scores', {}),
    "anomaly_score": anomaly_score,
    "graph_metrics": {
        "total_nodes": graph.number_of_nodes(),
        "total_edges": graph.number_of_edges(),
```

```
                "num_communities": len(communities)
            },
            "stylometric_features": current_features,
            "layer3_status": "completed",
            "timestamp_analyzed": datetime.utcnow().isoformat()
        }

        if result['flagged']:
            logger.warning(
                f"Coordination detected for {submission_id} "
                f"(confidence={result['confidence']:.3f}, "
                f"community_size={result['community_size']})"
            )
            if self.metrics:
                self.metrics.record_coordination_detection()
        else:
            logger.info(f"No coordination detected for {submission_id}")

        return result

    except Exception as e:
        logger.error(
            f"Layer 3 processing failed for {submission_id}: {e}",
            exc_info=True
        )
        raise ValueError(f"Coordination detection failed: {str(e)}")

def _get_recent_submissions(
    self,
    timestamp: Optional[datetime] = None
) -> List[Dict]:
    """
    Get submissions within time window.

    Args:
        timestamp: Reference timestamp

    Returns:
        list: Recent submission data
    """
    if timestamp is None:
        timestamp = datetime.utcnow()

    cutoff_time = timestamp - self.time_window

    # Get all submissions from storage
```

```
    all_submissions = self.storage.get_all_submissions()

    # Filter by time window
    recent = []
    for sub in all_submissions:
        sub_time_str = sub.get('timestamp_submission')
        if sub_time_str:
            try:
                sub_time = datetime.fromisoformat(sub_time_str.replace('Z', ''))
                if sub_time >= cutoff_time:
                    recent.append(sub)
            except Exception as e:
                logger.warning(f"Invalid timestamp format: {e}")

    return recent

def _extract_stylometric_features(
    self,
    submission_id: str,
    text: Optional[str]
) -> Dict:
    """
    Extract stylometric features from text.

    Features:
    - Type-Token Ratio (TTR)
    - Lexical density
    - Average word length
    - Sentence length statistics
    - POS tag distribution (simplified)

    Args:
        submission_id: Submission identifier
        text: Text to analyze

    Returns:
        dict: Stylometric features
    """
    if not text or len(text.strip()) < 10:
        # Return neutral features for short/missing text
        return {
            "ttr": 0.5,
            "lexical_density": 0.5,
            "avg_word_length": 5.0,
            "avg_sentence_length": 10.0,
            "text_length": 0,
```

```
            "has_text": False
        }

    try:
        # Use text_utils for feature extraction
        features = self.text_utils.extract_stylometric_features(text)
        features['has_text'] = True
        return features

    except Exception as e:
        logger.warning(f"Feature extraction failed: {e}")
        return {
            "ttr": 0.5,
            "lexical_density": 0.5,
            "avg_word_length": 5.0,
            "avg_sentence_length": 10.0,
            "text_length": len(text),
            "has_text": True
        }

def _build_coordination_graph(
    self,
    submissions: List[Dict],
    current_id: str,
    current_features: Dict,
    current_text: Optional[str]
) -> nx.Graph:
    """
    Build coordination graph with multiple edge types.

    Edge types:
    - style: Stylometric similarity
    - temporal: Submitted close in time
    - content: Semantic similarity

    Args:
        submissions: List of recent submissions
        current_id: Current submission ID
        current_features: Current submission features
        current_text: Current submission text

    Returns:
        nx.Graph: Coordination graph
    """
    G = nx.Graph()
```

```
# Add current submission as node
G.add_node(
    current_id,
    features=current_features,
    text=current_text,
    timestamp=datetime.utcnow()
)

# Add other submissions as nodes and compute edges
for sub in submissions:
    sub_id = sub.get('id')
    if not sub_id or sub_id == current_id:
        continue

    sub_text = sub.get('text_narrative', '')
    sub_features = sub.get('stylometric_features')

    # Extract features if not cached
    if not sub_features:
        sub_features = self._extract_stylometric_features(
            sub_id,
            sub_text
        )

    # Add node
    sub_time_str = sub.get('timestamp_submission')
    sub_time = datetime.utcnow()
    if sub_time_str:
        try:
            sub_time = datetime.fromisoformat(sub_time_str.replace('Z', ''))
        except Exception:
            pass

    G.add_node(
        sub_id,
        features=sub_features,
        text=sub_text,
        timestamp=sub_time
    )

    # Compute similarities
    style_sim = self._compute_style_similarity(
        current_features,
        sub_features
    )
```

```
            temporal_sim = self._compute_temporal_similarity(
                datetime.utcnow(),
                sub_time
            )

            content_sim = self._compute_content_similarity(
                current_text,
                sub_text
            )

            # Weighted overall similarity
            overall_sim = (
                0.4 * style_sim +
                0.3 * temporal_sim +
                0.3 * content_sim
            )

            # Add edge if similarity exceeds threshold
            if overall_sim >= self.min_similarity:
                G.add_edge(
                    current_id,
                    sub_id,
                    weight=overall_sim,
                    style=style_sim,
                    temporal=temporal_sim,
                    content=content_sim
                )
                logger.debug(
                    f"Edge added: {current_id} <-> {sub_id} "
                    f"(sim={overall_sim:.3f})"
                )

        logger.debug(
            f"Graph built: {G.number_of_nodes()} nodes, "
            f"{G.number_of_edges()} edges"
        )

        return G

    def _compute_style_similarity(
        self,
        features1: Dict,
        features2: Dict
    ) -> float:
        """
        Compute stylometric similarity between two submissions.
```

```
    Args:
        features1: Features of first submission
        features2: Features of second submission

    Returns:
        float: Similarity score [0, 1]
    """
    if not features1.get('has_text') or not features2.get('has_text'):
        return 0.0

    # Compare feature vectors
    keys = ['ttr', 'lexical_density', 'avg_word_length', 'avg_sentence_length']

    diffs = []
    for key in keys:
        val1 = features1.get(key, 0.5)
        val2 = features2.get(key, 0.5)

        # Normalize difference
        max_val = max(val1, val2) + 1e-6
        diff = abs(val1 - val2) / max_val
        diffs.append(diff)

    # Similarity = 1 - average difference
    similarity = 1.0 - np.mean(diffs)

    return max(0.0, min(1.0, similarity))

def _compute_temporal_similarity(
    self,
    time1: datetime,
    time2: datetime
) -> float:
    """
    Compute temporal similarity (submissions close in time).

    Args:
        time1: First timestamp
        time2: Second timestamp

    Returns:
        float: Similarity score [0, 1]
    """
    time_diff = abs((time1 - time2).total_seconds())
```

```
    # Exponential decay: similarity = exp(-diff / half_life)
    # half_life = 6 hours (submissions 6 hours apart have sim=0.5)
    half_life = 6 * 3600  # 6 hours in seconds

    similarity = np.exp(-time_diff / half_life)

    return float(similarity)

def _compute_content_similarity(
    self,
    text1: Optional[str],
    text2: Optional[str]
) -> float:
    """
    Compute semantic content similarity.

    Args:
        text1: First text
        text2: Second text

    Returns:
        float: Similarity score [0, 1]
    """
    if not text1 or not text2:
        return 0.0

    if len(text1) < 10 or len(text2) < 10:
        return 0.0

    try:
        # Use text_utils for similarity computation
        similarity = self.text_utils.compute_text_similarity(text1, text2)
        return similarity

    except Exception as e:
        logger.warning(f"Content similarity computation failed: {e}")
        return 0.0

def _detect_communities(self, graph: nx.Graph) -> List[Set[str]]:
    """
    Detect communities using Louvain algorithm.

    Args:
        graph: Coordination graph

    Returns:
```

```
        list: List of communities (sets of node IDs)
    """
    if graph.number_of_nodes() < 2:
        return []

    if graph.number_of_edges() == 0:
        # No edges = no communities
        return []

    try:
        # Use Louvain community detection
        communities_generator = community.louvain_communities(
            graph,
            seed=42,
            weight='weight'
        )

        communities = list(communities_generator)

        # Filter out single-node communities
        communities = [c for c in communities if len(c) >= 2]

        logger.debug(f"Detected {len(communities)} multi-node communities")

        return communities

    except Exception as e:
        logger.warning(f"Community detection failed: {e}")
        return []

def _check_coordination(
    self,
    submission_id: str,
    communities: List[Set[str]],
    graph: nx.Graph,
    current_features: Dict,
    all_submissions: List[Dict]
) -> Dict:
    """
    Check if submission is part of coordinated attack.

    Args:
        submission_id: Current submission ID
        communities: Detected communities
        graph: Coordination graph
        current_features: Current submission features
```

```
    all_submissions: All recent submissions

Returns:
    dict: Coordination check results
"""
# Find community containing current submission
current_community = None
community_id = None

for idx, comm in enumerate(communities):
    if submission_id in comm:
        current_community = comm
        community_id = idx
        break

if not current_community:
    # Not in any community
    return {
        'flagged': False,
        'confidence': 0.0,
        'community_id': None,
        'community_size': 0,
        'similarity_scores': {}
    }

community_size = len(current_community)

# Flag if community is large enough
flagged = community_size >= self.min_community_size

# Calculate confidence based on:
# 1. Community size
# 2. Edge weights (similarity)
# 3. Graph density

# Get edges within community
community_edges = []
for node1 in current_community:
    for node2 in current_community:
        if node1 < node2 and graph.has_edge(node1, node2):
            community_edges.append((node1, node2))

if community_edges:
    avg_similarity = np.mean([
        graph[u][v]['weight'] for u, v in community_edges
    ])
```

```
else:
    avg_similarity = 0.0

# Confidence calculation
# Factor 1: Community size (normalized)
size_factor = min(1.0, community_size / 10)

# Factor 2: Average similarity
sim_factor = avg_similarity

# Factor 3: Temporal clustering
if len(current_community) >= 2:
    timestamps = []
    for node_id in current_community:
        if graph.nodes[node_id].get('timestamp'):
            timestamps.append(graph.nodes[node_id]['timestamp'])

    if len(timestamps) >= 2:
        time_diffs = []
        timestamps_sorted = sorted(timestamps)
        for i in range(len(timestamps_sorted) - 1):
            diff = (timestamps_sorted[i+1] - timestamps_sorted[i]).total_seconds()
            time_diffs.append(diff)

        avg_time_diff = np.mean(time_diffs)
        # Closer in time = higher confidence
        temporal_factor = np.exp(-avg_time_diff / 3600)  # 1 hour half-life
    else:
        temporal_factor = 0.5
else:
    temporal_factor = 0.5

# Weighted confidence
confidence = (
    0.4 * size_factor +
    0.4 * sim_factor +
    0.2 * temporal_factor
)

confidence = max(0.0, min(1.0, confidence))

# Get similarity scores to community members
similarity_scores = {}
if graph.has_node(submission_id):
    for neighbor in graph.neighbors(submission_id):
        if neighbor in current_community:
```

```
                similarity_scores[neighbor] = graph[submission_id][neighbor]['weight']

    return {
        'flagged': flagged,
        'confidence': confidence,
        'community_id': community_id,
        'community_size': community_size,
        'similarity_scores': similarity_scores,
        'avg_similarity': avg_similarity,
        'temporal_clustering': temporal_factor
    }

def _detect_anomaly(
    self,
    current_features: Dict,
    recent_submissions: List[Dict]
) -> float:
    """
    Detect if submission is anomalous using One-Class SVM.

    Args:
        current_features: Current submission features
        recent_submissions: Recent submissions for training

    Returns:
        float: Anomaly score (higher = more anomalous)
    """
    # Need at least 10 samples to train
    if len(recent_submissions) < 10:
        return 0.0

    if not current_features.get('has_text'):
        return 0.0

    try:
        # Extract feature vectors
        feature_vectors = []
        for sub in recent_submissions:
            sub_features = sub.get('stylometric_features')
            if not sub_features:
                sub_text = sub.get('text_narrative', '')
                sub_features = self._extract_stylometric_features(
                    sub.get('id'),
                    sub_text
                )
```

```
            if sub_features.get('has_text'):
                vec = self._feature_dict_to_vector(sub_features)
                feature_vectors.append(vec)

        if len(feature_vectors) < 10:
            return 0.0

        # Train One-Class SVM
        X_train = np.array(feature_vectors)
        X_train = self.scaler.fit_transform(X_train)

        svm = OneClassSVM(kernel='rbf', gamma='auto', nu=0.1)
        svm.fit(X_train)

        # Predict on current submission
        current_vec = self._feature_dict_to_vector(current_features)
        current_vec = self.scaler.transform([current_vec])

        # Decision function: negative = anomaly
        decision = svm.decision_function(current_vec)[0]

        # Convert to [0, 1] score (higher = more anomalous)
        # Normalize decision function output
        anomaly_score = 1.0 / (1.0 + np.exp(decision))

        return float(anomaly_score)

    except Exception as e:
        logger.warning(f"Anomaly detection failed: {e}")
        return 0.0

def _feature_dict_to_vector(self, features: Dict) -> np.ndarray:
    """
    Convert feature dictionary to numpy vector.

    Args:
        features: Feature dictionary

    Returns:
        np.ndarray: Feature vector
    """
    keys = ['ttr', 'lexical_density', 'avg_word_length', 'avg_sentence_length']
    vector = [features.get(key, 0.5) for key in keys]
    return np.array(vector)

def get_coordination_graph_data(
```

```
    self,
    start_date: Optional[datetime] = None,
    end_date: Optional[datetime] = None,
    min_similarity: Optional[float] = None
) -> Dict:
    """
    Get coordination graph data for visualization.

    Args:
        start_date: Filter start date
        end_date: Filter end date
        min_similarity: Minimum edge similarity

    Returns:
        dict: Graph data with nodes, edges, communities
    """
    try:
        # Get submissions in date range
        all_submissions = self.storage.get_all_submissions()

        filtered_submissions = []
        for sub in all_submissions:
            sub_time_str = sub.get('timestamp_submission')
            if sub_time_str:
                try:
                    sub_time = datetime.fromisoformat(sub_time_str.replace('Z', ''))

                    if start_date and sub_time < start_date:
                        continue
                    if end_date and sub_time > end_date:
                        continue

                    filtered_submissions.append(sub)
                except Exception:
                    pass

        # Build graph with all filtered submissions
        G = nx.Graph()

        for sub in filtered_submissions:
            sub_id = sub.get('id')
            G.add_node(
                sub_id,
                pseudonym=sub.get('pseudonym', 'N/A'),
                score=sub.get('credibility', {}).get('final_score', 0.5),
                flagged=sub.get('coordination', {}).get('flagged', False)
```

```
    )

# Add edges based on cached coordination data
threshold = min_similarity if min_similarity else self.min_similarity

for sub in filtered_submissions:
    sub_id = sub.get('id')
    coord_data = sub.get('coordination', {})
    similarity_scores = coord_data.get('similarity_scores', {})

    for other_id, sim in similarity_scores.items():
        if sim >= threshold and G.has_node(other_id):
            if not G.has_edge(sub_id, other_id):
                G.add_edge(sub_id, other_id, weight=sim)

# Detect communities
communities = self._detect_communities(G)

# Convert to serializable format
nodes = []
for node_id in G.nodes():
    node_data = G.nodes[node_id]
    nodes.append({
        'id': node_id,
        'label': node_data.get('pseudonym', node_id[:8]),
        'score': node_data.get('score', 0.5),
        'flagged': node_data.get('flagged', False)
    })

edges = []
for u, v in G.edges():
    edges.append({
        'source': u,
        'target': v,
        'weight': G[u][v]['weight']
    })

community_data = []
for idx, comm in enumerate(communities):
    community_data.append({
        'id': idx,
        'members': list(comm),
        'size': len(comm)
    })

return {
```

```
                'nodes': nodes,
                'edges': edges,
                'communities': community_data,
                'total_submissions': len(nodes),
                'flagged_submissions': sum(1 for n in nodes if n['flagged'])
            }

        except Exception as e:
            logger.error(f"Failed to get graph data: {e}")
            return {
                'nodes': [],
                'edges': [],
                'communities': [],
                'total_submissions': 0,
                'flagged_submissions': 0
            }


"""from backend.utils.text_utils import TextUtils, extract_features
from backend.utils.graph_utils import GraphUtils, detect_coordination

class CoordinationDetectionLayer:
    def analyze_submissions(self, submissions: List[Dict]):

        # Extract stylometric features
        for submission in submissions:
            if 'narrative' in submission:
                features = extract_features(submission['narrative'])
                submission['stylometric_features'] = features

        # Create graph
        G = GraphUtils.create_submission_graph(
            submissions,
            similarity_threshold=0.3
        )

        # Detect communities and patterns
        communities, patterns = detect_coordination(G)

        # Calculate graph metrics
        metrics = GraphUtils.calculate_graph_metrics(G)

        return {
            'graph': G,
            'communities': communities,
            'suspicious_patterns': patterns,
```

```
            'metrics': metrics
        }
"""
"""
Layer 4: Byzantine Consensus Simulator
Simulates distributed validators with Byzantine fault tolerance.

Input: Submission with credibility and coordination flags
Output: Consensus decision and vote distribution
"""

import logging
import random
from datetime import datetime
from enum import Enum
from typing import Dict, List, Optional, Tuple

import numpy as np

from backend.services.metrics_service import MetricsService


# Initialize logger
logger = logging.getLogger(__name__)


class DecisionType(str, Enum):
    """Consensus decision types."""
    FORWARD = "forward"  # Forward to authorities
    REVIEW = "review"    # Requires human review
    REJECT = "reject"    # Insufficient evidence


class ValidatorType(str, Enum):
    """Validator behavior types."""
    HONEST = "honest"
    DEVILS_ADVOCATE = "devils_advocate"
    RANDOM = "random"


class Layer4Consensus:
    """
    Layer 4: Byzantine Consensus Simulator

    Implements:
    - Simulated validator pool (15-20 validators)
```

```
- Devil's advocate validators (10%)
- Weighted voting based on validator reputation
- Adaptive fault tolerance
- Majority voting with thresholds
"""

def __init__(
    self,
    storage_service,
    num_validators: int = 17,
    devils_advocate_ratio: float = 0.1,
    consensus_threshold: float = 0.67,
    forward_threshold: float = 0.75,
    reject_threshold: float = 0.30,
    metrics_service: Optional[MetricsService] = None
):
    """
    Initialize Layer 4 with validator configuration.

    Args:
        storage_service: Storage service for validator state
        num_validators: Total number of validators
        devils_advocate_ratio: Ratio of devil's advocate validators
        consensus_threshold: Minimum agreement for consensus
        forward_threshold: Score threshold for forwarding
        reject_threshold: Score threshold for rejection
    """
    self.storage = storage_service
    self.num_validators = num_validators
    self.devils_advocate_ratio = devils_advocate_ratio
    self.consensus_threshold = consensus_threshold
    self.forward_threshold = forward_threshold
    self.reject_threshold = reject_threshold
    self.metrics = metrics_service
    # Initialize or load validators
    self.validators = self._initialize_validators()

    logger.info(
        f"Layer 4 (Consensus) initialized with {num_validators} validators "
        f"({int(devils_advocate_ratio*100)}% devil's advocate)"
    )

def process(
    self,
    submission_id: str,
    credibility_score: float,
```

```
    coordination_flagged: bool,
    coordination_confidence: float = 0.0
) -> Dict:
    """
    Process submission through Byzantine consensus.

    Args:
        submission_id: Unique submission identifier
        credibility_score: Final credibility score from Layer 2
        coordination_flagged: Whether coordination was detected
        coordination_confidence: Coordination detection confidence

    Returns:
        dict: Consensus results

    Raises:
        ValueError: If consensus fails
    """
    logger.info(f"Layer 4 processing submission {submission_id}")

    try:
        # Step 1: Adjust score based on coordination flag
        adjusted_score = self._adjust_score_for_coordination(
            credibility_score,
            coordination_flagged,
            coordination_confidence
        )

        # Step 2: Get validator votes
        votes = self._collect_votes(
            submission_id,
            adjusted_score,
            coordination_flagged
        )

        # Step 3: Aggregate votes with weights
        vote_counts, weighted_scores = self._aggregate_votes(votes)

        # Step 4: Determine consensus decision
        decision, agreement = self._determine_decision(
            adjusted_score,
            vote_counts,
            weighted_scores
        )

        # Step 5: Update validator reputations
```

```
            self._update_validator_reputations(votes, decision)

            result = {
                "submission_id": submission_id,
                "decision": decision.value,
                "adjusted_score": adjusted_score,
                "original_score": credibility_score,
                "votes": vote_counts,
                "validator_scores": [v['score'] for v in votes],
                "agreement_percentage": agreement * 100,
                "consensus_reached": agreement >= self.consensus_threshold,
                "num_validators": len(votes),
                "layer4_status": "completed",
                "timestamp_consensus": datetime.utcnow().isoformat()
            }

            logger.info(
                f"Layer 4 completed for {submission_id}: "
                f"decision={decision.value}, agreement={agreement:.1%}"
            )

            return result

        except Exception as e:
            logger.error(
                f"Layer 4 processing failed for {submission_id}: {e}",
                exc_info=True
            )
            raise ValueError(f"Consensus processing failed: {str(e)}")

    def _initialize_validators(self) -> List[Dict]:
        """
        Initialize or load validator pool.

        Returns:
            list: Validator configurations
        """
        try:
            # Try to load existing validators
            validators = self.storage.load_validators()

            if validators and len(validators) == self.num_validators:
                logger.debug(f"Loaded {len(validators)} existing validators")
                return validators
        except Exception:
            pass
```

```
    # Create new validators
    logger.info(f"Creating {self.num_validators} new validators")

    validators = []
    num_devils_advocate = int(self.num_validators * self.devils_advocate_ratio)

    for i in range(self.num_validators):
        # Determine validator type
        if i < num_devils_advocate:
            v_type = ValidatorType.DEVILS_ADVOCATE
        else:
            v_type = ValidatorType.HONEST

        validator = {
            "id": f"validator_{i+1:02d}",
            "type": v_type.value,
            "reputation": 1.0,  # Start with perfect reputation
            "total_votes": 0,
            "correct_votes": 0,
            "bias": random.uniform(-0.1, 0.1)  # Small random bias
        }
        validators.append(validator)

    # Save validators
    try:
        self.storage.save_validators(validators)
    except Exception as e:
        logger.warning(f"Failed to save validators: {e}")

    return validators

def _adjust_score_for_coordination(
    self,
    credibility_score: float,
    coordination_flagged: bool,
    coordination_confidence: float
) -> float:
    """
    Adjust credibility score if coordination detected.

    Args:
        credibility_score: Original credibility score
        coordination_flagged: Whether coordination detected
        coordination_confidence: Detection confidence
```

```
        Returns:
            float: Adjusted score
        """
        if not coordination_flagged:
            return credibility_score

        # Reduce score based on coordination confidence
        penalty = 0.3 * coordination_confidence  # Up to 30% reduction
        adjusted_score = credibility_score * (1.0 - penalty)

        logger.debug(
            f"Score adjusted for coordination: "
            f"{credibility_score:.3f} !' {adjusted_score:.3f}"
        )

        return max(0.0, adjusted_score)

    def _collect_votes(
        self,
        submission_id: str,
        score: float,
        coordination_flagged: bool
    ) -> List[Dict]:
        """
        Collect votes from all validators.

        Args:
            submission_id: Submission identifier
            score: Adjusted credibility score
            coordination_flagged: Coordination flag

        Returns:
            list: Validator votes
        """
        votes = []

        for validator in self.validators:
            # Simulate validator's score assessment
            validator_score = self._simulate_validator_vote(
                validator,
                score,
                coordination_flagged
            )

            # Determine vote (accept/reject)
            vote = "accept" if validator_score >= 0.5 else "reject"
```

```
            votes.append({
                "validator_id": validator['id'],
                "validator_type": validator['type'],
                "score": validator_score,
                "vote": vote,
                "weight": validator['reputation']
            })

        return votes

    def _simulate_validator_vote(
        self,
        validator: Dict,
        score: float,
        coordination_flagged: bool
    ) -> float:
        """
        Simulate how a validator would score the submission.

        Args:
            validator: Validator configuration
            score: Credibility score
            coordination_flagged: Coordination flag

        Returns:
            float: Validator's score assessment
        """
        validator_type = validator['type']
        bias = validator['bias']

        if validator_type == ValidatorType.HONEST.value:
            # Honest validator: small random noise + bias
            noise = random.gauss(0, 0.05)
            validator_score = score + bias + noise

        elif validator_type == ValidatorType.DEVILS_ADVOCATE.value:
            # Devil's advocate: systematically skeptical
            # Reduces score, especially for borderline cases
            if 0.4 <= score <= 0.7:
                # Most skeptical in uncertain range
                reduction = random.uniform(0.15, 0.25)
            else:
                reduction = random.uniform(0.05, 0.15)

            validator_score = score - reduction
```

```
        else:  # RANDOM
            # Random validator: uniform random score
            validator_score = random.uniform(0.0, 1.0)

        # Additional penalty for coordination-flagged submissions
        if coordination_flagged:
            validator_score *= random.uniform(0.8, 0.95)

        # Clip to [0, 1]
        return max(0.0, min(1.0, validator_score))

    def _aggregate_votes(
        self,
        votes: List[Dict]
    ) -> Tuple[Dict[str, int], List[float]]:
        """
        Aggregate validator votes with reputation weighting.

        Args:
            votes: List of validator votes

        Returns:
            tuple: (vote_counts, weighted_scores)
        """
        # Count votes
        vote_counts = {"accept": 0, "reject": 0}

        # Weighted scores
        weighted_scores = []
        total_weight = 0.0

        for vote in votes:
            vote_counts[vote['vote']] += 1

            weight = vote['weight']
            weighted_scores.append(vote['score'] * weight)
            total_weight += weight

        # Normalize weighted scores
        if total_weight > 0:
            weighted_scores = [s / total_weight for s in weighted_scores]

        return vote_counts, weighted_scores

    def _determine_decision(
```

```
    self,
    score: float,
    vote_counts: Dict[str, int],
    weighted_scores: List[float]
) -> Tuple[DecisionType, float]:
    """
    Determine consensus decision based on votes and thresholds.

    Args:
        score: Adjusted credibility score
        vote_counts: Vote counts by type
        weighted_scores: Weighted validator scores

    Returns:
        tuple: (decision, agreement_percentage)
    """
    total_votes = sum(vote_counts.values())

    if total_votes == 0:
        return DecisionType.REVIEW, 0.0

    # Calculate agreement (majority vote percentage)
    max_votes = max(vote_counts.values())
    agreement = max_votes / total_votes

    # Determine decision based on score and thresholds
    if score >= self.forward_threshold and vote_counts['accept'] > vote_counts['reject']:
        decision = DecisionType.FORWARD
    elif score <= self.reject_threshold and vote_counts['reject'] > vote_counts['accept']:
        decision = DecisionType.REJECT
    else:
        # Borderline cases require human review
        decision = DecisionType.REVIEW

    # Override if consensus not reached
    if agreement < self.consensus_threshold:
        decision = DecisionType.REVIEW
        logger.warning(
            f"Consensus threshold not met (agreement={agreement:.1%}), "
            f"escalating to human review"
        )

    return decision, agreement

def _update_validator_reputations(
    self,
```

```
    votes: List[Dict],
    decision: DecisionType
) -> None:
    """
    Update validator reputations based on consensus outcome.

    Validators who voted with the majority gain reputation.
    Devil's advocates maintain reputation for their role.

    Args:
        votes: Validator votes
        decision: Final consensus decision

    Returns:
        None
    """
    # Determine majority vote
    vote_counts = {"accept": 0, "reject": 0}
    for vote in votes:
        vote_counts[vote['vote']] += 1

    majority_vote = "accept" if vote_counts['accept'] > vote_counts['reject'] else "reject"

    # Update reputations
    for i, vote in enumerate(votes):
        validator = self.validators[i]

        validator['total_votes'] += 1

        # Check if validator voted with majority
        voted_with_majority = (vote['vote'] == majority_vote)

        if voted_with_majority:
            validator['correct_votes'] += 1
            # Increase reputation slightly
            validator['reputation'] = min(1.0, validator['reputation'] + 0.01)
        else:
            # Decrease reputation slightly (but not for devil's advocates)
            if validator['type'] != ValidatorType.DEVILS_ADVOCATE.value:
                validator['reputation'] = max(0.1, validator['reputation'] - 0.01)

    # Save updated validators
    try:
        self.storage.save_validators(self.validators)
    except Exception as e:
        logger.warning(f"Failed to save validator updates: {e}")
```

```
    def get_validator_statistics(self) -> Dict:
        """
        Get statistics about validator pool.

        Returns:
            dict: Validator statistics
        """
        stats = {
            "total_validators": len(self.validators),
            "avg_reputation": np.mean([v['reputation'] for v in self.validators]),
            "validator_types": {},
            "top_validators": []
        }

        # Count by type
        for v in self.validators:
            v_type = v['type']
            stats['validator_types'][v_type] = stats['validator_types'].get(v_type, 0) + 1

        # Top 5 validators by reputation
        sorted_validators = sorted(
            self.validators,
            key=lambda v: v['reputation'],
            reverse=True
        )

        stats['top_validators'] = [
            {
                "id": v['id'],
                "reputation": v['reputation'],
                "accuracy": v['correct_votes'] / v['total_votes'] if v['total_votes'] > 0 else 0.0
            }
            for v in sorted_validators[:5]
        ]

        return stats

"""
Layer 5: Counter-Evidence Processor
Bayesian aggregation with presumption-of-innocence weighting.

Input: Original submission + counter-evidence
Output: Updated posterior scores and decision
"""
```

```
import logging
from datetime import datetime
from typing import Dict, Optional, Tuple

import numpy as np
from scipy import stats

from backend.services.metrics_service import MetricsService


# Initialize logger
logger = logging.getLogger(__name__)


class Layer5CounterEvidence:
    """
    Layer 5: Counter-Evidence Processor

    Implements:
    - Bayesian aggregation of accusation and defense
    - 1.3× presumption-of-innocence weighting
    - 1.2× identity verification bonus
    - Posterior probability calculation
    - Decision update logic
    """

    def __init__(
        self,
        storage_service,
        presumption_weight: float = 1.3,
        identity_bonus: float = 1.2,
        decision_change_threshold: float = 0.15,
        metrics_service: Optional[MetricsService] = None
    ):
        """
        Initialize Layer 5 with configuration.

        Args:
            storage_service: Storage service for submissions
            presumption_weight: Presumption of innocence multiplier
            identity_bonus: Identity verification bonus multiplier
            decision_change_threshold: Minimum score change to update decision
        """
        self.storage = storage_service
        self.presumption_weight = presumption_weight
        self.identity_bonus = identity_bonus
```

```
        self.decision_change_threshold = decision_change_threshold
        self.metrics = metrics_service
        logger.info(
            f"Layer 5 (Counter-Evidence) initialized "
            f"(presumption={presumption_weight}×, identity={identity_bonus}×)"
        )

    def process(
        self,
        original_submission_id: str,
        counter_evidence_id: str,
        counter_credibility_score: float,
        identity_verified: bool = False
    ) -> Dict:
        """
        Process counter-evidence and update original submission.

        Args:
            original_submission_id: Original submission ID
            counter_evidence_id: Counter-evidence submission ID
            counter_credibility_score: Counter-evidence credibility score
            identity_verified: Whether defense verified identity

        Returns:
            dict: Aggregation results with updated scores and decision

        Raises:
            ValueError: If processing fails
        """
        logger.info(
            f"Layer 5 processing counter-evidence {counter_evidence_id} "
            f"for {original_submission_id}"
        )

        try:
            # Step 1: Load original submission
            original = self.storage.load_submission(original_submission_id)
            if not original:
                raise ValueError(f"Original submission {original_submission_id} not found")

            # Step 2: Extract original scores
            original_credibility = original.get('credibility', {})
            original_score = original_credibility.get('final_score', 0.5)
            original_decision = original.get('consensus', {}).get('decision', 'review')

            logger.debug(
```

```
    f"Original: score={original_score:.3f}, decision={original_decision}"
)

# Step 3: Apply Bayesian aggregation
posterior_score, likelihood_ratio = self._bayesian_aggregation(
    accusation_score=original_score,
    defense_score=counter_credibility_score,
    identity_verified=identity_verified
)

# Step 4: Calculate score change
score_delta = posterior_score - original_score

# Step 5: Determine if decision should change
decision_changed, new_decision = self._evaluate_decision_change(
    original_score=original_score,
    posterior_score=posterior_score,
    score_delta=score_delta,
    original_decision=original_decision
)

# Step 6: Calculate confidence intervals
posterior_ci = self._calculate_posterior_ci(
    original_score=original_score,
    counter_score=counter_credibility_score,
    posterior_score=posterior_score
)

result = {
    "original_submission_id": original_submission_id,
    "counter_evidence_id": counter_evidence_id,
    "original_score": original_score,
    "counter_score": counter_credibility_score,
    "posterior_score": posterior_score,
    "score_delta": score_delta,
    "likelihood_ratio": likelihood_ratio,
    "identity_verified": identity_verified,
    "identity_bonus_applied": identity_verified,
    "presumption_weight_applied": self.presumption_weight,
    "decision_changed": decision_changed,
    "original_decision": original_decision,
    "new_decision": new_decision,
    "posterior_confidence_interval": posterior_ci,
    "aggregation_method": "bayesian",
    "layer5_status": "completed",
    "timestamp_aggregated": datetime.utcnow().isoformat()
```

```
            }

            if decision_changed:
                logger.warning(
                    f"Decision changed for {original_submission_id}: "
                    f"{original_decision} !' {new_decision} "
                    f"(9G66÷&S×·66÷&UöFVÇF ¢²ã6gÒ'
                )
            else:
                logger.info(
                    f"Decision unchanged for {original_submission_id}: "
                    f"{new_decision} (9G66÷&S×·66÷&UöFVÇF ¢²ã6gÒ'
                )

            return result

        except Exception as e:
            logger.error(
                f"Layer 5 processing failed: {e}",
                exc_info=True
            )
            raise ValueError(f"Counter-evidence processing failed: {str(e)}")

    def _bayesian_aggregation(
        self,
        accusation_score: float,
        defense_score: float,
        identity_verified: bool
    ) -> Tuple[float, float]:
        """
        Perform Bayesian aggregation of accusation and defense.

        Uses Bayes' theorem with presumption-of-innocence weighting:
        P(guilty|evidence) = P(evidence|guilty) * P(guilty) / P(evidence)

        Args:
            accusation_score: Original accusation credibility score
            defense_score: Counter-evidence credibility score
            identity_verified: Whether defense verified identity

        Returns:
            tuple: (posterior_score, likelihood_ratio)
        """
        # Convert scores to probabilities (already in [0,1])
        p_accusation = accusation_score
        p_defense = defense_score
```

```
# Apply identity verification bonus to defense
if identity_verified:
    p_defense = min(1.0, p_defense * self.identity_bonus)
    logger.debug(
        f"Identity bonus applied: {defense_score:.3f} !' {p_defense:.3f}"
    )

# Apply presumption of innocence (weight defense more heavily)
weighted_defense = p_defense * self.presumption_weight

# Bayesian update: combine accusation and weighted defense
# Likelihood ratio approach

# P(guilty | accusation)
prior_guilty = p_accusation
prior_innocent = 1.0 - p_accusation

# P(defense evidence | guilty) vs P(defense evidence | innocent)
# If defense score is high, it's more likely the person is innocent
# Use defense score as P(defense | innocent)
p_evidence_if_innocent = weighted_defense
p_evidence_if_guilty = 1.0 - weighted_defense

# Likelihood ratio
likelihood_ratio = p_evidence_if_innocent / (p_evidence_if_guilty + 1e-10)

# Posterior using Bayes' theorem
numerator = p_evidence_if_innocent * prior_innocent
denominator = (p_evidence_if_innocent * prior_innocent +
               p_evidence_if_guilty * prior_guilty)

if denominator < 1e-10:
    posterior_innocent = 0.5
else:
    posterior_innocent = numerator / denominator

# Posterior probability of guilt (what we report)
posterior_guilty = 1.0 - posterior_innocent

# Clip to [0, 1]
posterior_score = max(0.0, min(1.0, posterior_guilty))

logger.debug(
    f"Bayesian aggregation: "
    f"P(guilt)={p_accusation:.3f}, "
```

```
            f"P(defense|innocent)={weighted_defense:.3f}, "
            f"P(guilt|defense)={posterior_score:.3f}, "
            f"LR={likelihood_ratio:.3f}"
        )

        return posterior_score, likelihood_ratio

    def _evaluate_decision_change(
        self,
        original_score: float,
        posterior_score: float,
        score_delta: float,
        original_decision: str
    ) -> Tuple[bool, str]:
        """
        Evaluate if decision should change based on posterior score.

        Args:
            original_score: Original credibility score
            posterior_score: Posterior score after counter-evidence
            score_delta: Change in score
            original_decision: Original consensus decision

        Returns:
            tuple: (decision_changed, new_decision)
        """
        # Decision thresholds (from Layer 4)
        FORWARD_THRESHOLD = 0.75
        REJECT_THRESHOLD = 0.30

        # Determine new decision based on posterior score
        if posterior_score >= FORWARD_THRESHOLD:
            new_decision = "forward"
        elif posterior_score <= REJECT_THRESHOLD:
            new_decision = "reject"
        else:
            new_decision = "review"

        # Check if decision actually changed
        decision_changed = (new_decision != original_decision)

        # Additional check: only change if delta is significant
        if decision_changed and abs(score_delta) < self.decision_change_threshold:
            # Delta too small, keep original decision
            new_decision = original_decision
            decision_changed = False
```

```
            logger.debug(
                f"Decision change rejected: delta ({score_delta:.3f}) "
                f"below threshold ({self.decision_change_threshold})"
            )

        return decision_changed, new_decision

    def _calculate_posterior_ci(
        self,
        original_score: float,
        counter_score: float,
        posterior_score: float,
        confidence: float = 0.90
    ) -> Tuple[float, float]:
        """
        Calculate confidence interval for posterior score.

        Args:
            original_score: Original score
            counter_score: Counter-evidence score
            posterior_score: Posterior score
            confidence: Confidence level

        Returns:
            tuple: (lower_bound, upper_bound)
        """
        # Estimate variance from score uncertainty
        # Assume both scores have some inherent uncertainty

        # Variance estimation: use distance from 0.5 as indicator
        # Scores near 0.5 have higher uncertainty
        original_variance = 0.01 + 0.05 * (1 - 2 * abs(original_score - 0.5))
        counter_variance = 0.01 + 0.05 * (1 - 2 * abs(counter_score - 0.5))

        # Combined variance (assuming independence)
        combined_variance = (original_variance + counter_variance) / 2
        std_error = np.sqrt(combined_variance)

        # Calculate CI using normal approximation
        z_score = stats.norm.ppf((1 + confidence) / 2)
        margin = z_score * std_error

        lower = max(0.0, posterior_score - margin)
        upper = min(1.0, posterior_score + margin)

        return (float(lower), float(upper))
```

```
def generate_comparison_report(
    self,
    original_submission_id: str,
    counter_evidence_id: str
) -> Dict:
    """
    Generate before/after comparison report.

    Args:
        original_submission_id: Original submission ID
        counter_evidence_id: Counter-evidence ID

    Returns:
        dict: Comparison report data
    """
    try:
        # Load both submissions
        original = self.storage.load_submission(original_submission_id)
        counter = self.storage.load_submission(counter_evidence_id)

        if not original or not counter:
            raise ValueError("Submission not found")

        # Extract key metrics
        original_cred = original.get('credibility', {})
        counter_cred = counter.get('credibility', {})

        comparison = {
            "original": {
                "submission_id": original_submission_id,
                "credibility_score": original_cred.get('final_score', 0.0),
                "decision": original.get('consensus', {}).get('decision', 'unknown'),
                "coordination_flagged": original.get('coordination', {}).get('flagged', False),
                "timestamp": original.get('timestamp_submission')
            },
            "counter_evidence": {
                "submission_id": counter_evidence_id,
                "credibility_score": counter_cred.get('final_score', 0.0),
                "identity_verified": counter.get('verified_identity', False),
                "timestamp": counter.get('timestamp_submission')
            },
            "aggregated": {
                "posterior_score": original.get('posterior_score'),
                "score_delta": original.get('score_delta'),
                "new_decision": original.get('new_decision'),
```

```
                "decision_changed": original.get('decision_changed', False)
            },
            "generated_at": datetime.utcnow().isoformat()
        }

        return comparison

    except Exception as e:
        logger.error(f"Failed to generate comparison report: {e}")
        return {}

def calculate_impact_metrics(
    self,
    original_score: float,
    posterior_score: float,
    likelihood_ratio: float
) -> Dict:
    """
    Calculate impact metrics for research analysis.

    Args:
        original_score: Original credibility score
        posterior_score: Posterior score
        likelihood_ratio: Bayesian likelihood ratio

    Returns:
        dict: Impact metrics
    """
    score_change_percent = ((posterior_score - original_score) /
                            (original_score + 1e-10)) * 100

    absolute_change = abs(posterior_score - original_score)

    # Classify impact level
    if absolute_change < 0.1:
        impact_level = "minimal"
    elif absolute_change < 0.2:
        impact_level = "moderate"
    elif absolute_change < 0.3:
        impact_level = "significant"
    else:
        impact_level = "major"

    # Bayes factor interpretation
    if likelihood_ratio > 10:
        evidence_strength = "strong support for innocence"
```

```
        elif likelihood_ratio > 3:
            evidence_strength = "moderate support for innocence"
        elif likelihood_ratio > 1:
            evidence_strength = "weak support for innocence"
        elif likelihood_ratio > 0.33:
            evidence_strength = "weak support for guilt"
        elif likelihood_ratio > 0.1:
            evidence_strength = "moderate support for guilt"
        else:
            evidence_strength = "strong support for guilt"

        return {
            "score_change_percent": score_change_percent,
            "absolute_change": absolute_change,
            "impact_level": impact_level,
            "likelihood_ratio": likelihood_ratio,
            "evidence_strength": evidence_strength,
            "false_positive_reduction": max(0, original_score - posterior_score) > 0.2
        }

"""from backend.utils.math_utils import MathUtils, bayesian_aggregate
from backend.utils.time_utils import TimeUtils, now

class CounterEvidenceLayer:
    PRESUMPTION_OF_INNOCENCE = 1.3
    IDENTITY_VERIFICATION_BONUS = 1.2

    def process_counter_evidence(self, submission_id: str, counter_evidence: dict):

        # Get original submission
        original = self.storage.get_submission(submission_id)
        original_score = original.get('credibility_score', 0.5)

        # Process counter-evidence
        counter_score = counter_evidence.get('credibility_score', 0.5)
        is_verified = counter_evidence.get('identity_verified', False)

        # Apply weights
        counter_weight = self.PRESUMPTION_OF_INNOCENCE
        if is_verified:
            counter_weight *= self.IDENTITY_VERIFICATION_BONUS

        # Bayesian update
        prior = original_score
        likelihood = 1 - counter_score  # Inverse for counter-evidence
```

```
        new_score = MathUtils.bayesian_update(prior, likelihood, counter_weight)

        # Calculate confidence
        confidence = MathUtils.calculate_confidence_score(
            new_score,
            num_samples=2,  # Original + counter
            min_samples=3
        )

        return {
            'submission_id': submission_id,
            'original_score': original_score,
            'counter_score': counter_score,
            'updated_score': new_score,
            'confidence': confidence,
            'timestamp': now()
        }
"""
"""
Layer 6: Forensic Report Generator
Generates Section 45-compliant PDF reports with visualizations.

Input: Submission data with all assessment results
Output: PDF forensic report
"""

import io
import logging
from datetime import datetime
from pathlib import Path
from typing import Dict, List, Optional

import matplotlib
matplotlib.use('Agg')  # Non-interactive backend
import matplotlib.pyplot as plt
import numpy as np
from reportlab.lib import colors
from reportlab.lib.pagesizes import letter, A4
from reportlab.lib.styles import getSampleStyleSheet, ParagraphStyle
from reportlab.lib.units import inch
from reportlab.platypus import (
    SimpleDocTemplate, Table, TableStyle, Paragraph, Spacer,
    PageBreak, Image as RLImage, KeepTogether
)
from reportlab.lib.enums import TA_CENTER, TA_JUSTIFY, TA_LEFT
```

```
from backend.services.metrics_service import MetricsService

# Initialize logger
logger = logging.getLogger(__name__)


class Layer6Reporting:
    """
    Layer 6: Forensic Report Generator

    Implements:
    - Section 45-compliant PDF reports
    - Chain-of-custody documentation
    - Credibility score visualization
    - Confidence interval charts
    - Attention heatmaps (simplified for MVP)
    - Evidence metadata summary
    """

    def __init__(
        self,
        storage_service,
        hash_chain_service,
        output_dir: Optional[Path] = None,
        metrics_service: Optional[MetricsService] = None
    ):
        """
        Initialize Layer 6 with services.

        Args:
            storage_service: Storage service for submissions
            hash_chain_service: Hash chain service for custody proof
            output_dir: Output directory for reports
        """
        self.storage = storage_service
        self.hash_chain = hash_chain_service

        if output_dir:
            self.output_dir = Path(output_dir)
        else:
            self.output_dir = Path("backend/data/reports")

        self.output_dir.mkdir(parents=True, exist_ok=True)
        self.metrics = metrics_service
        logger.info(f"Layer 6 (Reporting) initialized (output: {self.output_dir})")
```

```
    def generate_report(
        self,
        submission_id: str,
        include_technical_details: bool = True
    ) -> Path:
        """
        Generate comprehensive forensic report.

        Args:
            submission_id: Submission identifier
            include_technical_details: Include technical analysis details

        Returns:
            Path: Path to generated PDF report

        Raises:
            ValueError: If report generation fails
        """
        logger.info(f"Layer 6 generating report for {submission_id}")

        try:
            # Step 1: Load submission data
            submission = self.storage.load_submission(submission_id)
            if not submission:
                raise ValueError(f"Submission {submission_id} not found")

            # Step 2: Create PDF document
            report_filename = f"forensic_report_{submission_id[:8]}
_{datetime.utcnow().strftime('%Y%m%d_%H%M%S')}.pdf"
            report_path = self.output_dir / report_filename

            doc = SimpleDocTemplate(
                str(report_path),
                pagesize=letter,
                rightMargin=0.75*inch,
                leftMargin=0.75*inch,
                topMargin=1*inch,
                bottomMargin=0.75*inch
            )

            # Step 3: Build report content
            story = []
            styles = self._get_custom_styles()

            # Title page
            story.extend(self._build_title_page(submission, styles))
```

```
        story.append(PageBreak())

        # Executive summary
        story.extend(self._build_executive_summary(submission, styles))
        story.append(Spacer(1, 0.3*inch))

        # Chain of custody
        story.extend(self._build_chain_of_custody(submission, styles))
        story.append(Spacer(1, 0.3*inch))

        # Credibility assessment
        story.extend(self._build_credibility_section(submission, styles))
        story.append(Spacer(1, 0.3*inch))

        # Coordination analysis
        story.extend(self._build_coordination_section(submission, styles))
        story.append(Spacer(1, 0.3*inch))

        # Consensus results
        story.extend(self._build_consensus_section(submission, styles))

        # Counter-evidence (if exists)
        if submission.get('counter_evidence_id'):
            story.append(PageBreak())
            story.extend(self._build_counter_evidence_section(submission, styles))

        # Technical details (optional)
        if include_technical_details:
            story.append(PageBreak())
            story.extend(self._build_technical_details(submission, styles))

        # Legal disclaimer
        story.append(PageBreak())
        story.extend(self._build_legal_disclaimer(styles))

        # Step 4: Build PDF
        doc.build(story)

        logger.info(f"Report generated: {report_path}")

        return report_path

    except Exception as e:
        logger.error(f"Report generation failed: {e}", exc_info=True)
        raise ValueError(f"Report generation failed: {str(e)}")
```

```
def _get_custom_styles(self) -> Dict:
    """Get custom paragraph styles."""
    styles = getSampleStyleSheet()

    # Custom styles
    styles.add(ParagraphStyle(
        name='CustomTitle',
        parent=styles['Heading1'],
        fontSize=24,
        textColor=colors.HexColor('#1a1a1a'),
        spaceAfter=30,
        alignment=TA_CENTER,
        fontName='Helvetica-Bold'
    ))

    styles.add(ParagraphStyle(
        name='SectionHeader',
        parent=styles['Heading2'],
        fontSize=14,
        textColor=colors.HexColor('#2c3e50'),
        spaceAfter=12,
        spaceBefore=12,
        fontName='Helvetica-Bold'
    ))

    styles.add(ParagraphStyle(
        name='BodyJustify',
        parent=styles['BodyText'],
        fontSize=10,
        alignment=TA_JUSTIFY,
        spaceAfter=12
    ))

    return styles

def _build_title_page(
    self,
    submission: Dict,
    styles: Dict
) -> List:
    """Build report title page."""
    elements = []

    # Title
    elements.append(Spacer(1, 1.5*inch))
    elements.append(Paragraph(
```

```
            "FORENSIC EVIDENCE ASSESSMENT REPORT",
            styles['CustomTitle']
        ))

        elements.append(Spacer(1, 0.5*inch))

        # Submission info table
        info_data = [
            ['Report ID:', submission.get('id', 'N/A')[:16]],
            ['Pseudonym:', submission.get('pseudonym', 'N/A')],
            ['Generated:', datetime.utcnow().strftime('%Y-%m-%d %H:%M:%S UTC')],
            ['Status:', submission.get('status', 'unknown').upper()],
            ['Classification:', self._get_classification_label(submission)]
        ]

        info_table = Table(info_data, colWidths=[2*inch, 4*inch])
        info_table.setStyle(TableStyle([
            ('FONTNAME', (0, 0), (-1, -1), 'Helvetica'),
            ('FONTSIZE', (0, 0), (-1, -1), 10),
            ('FONTNAME', (0, 0), (0, -1), 'Helvetica-Bold'),
            ('TEXTCOLOR', (0, 0), (0, -1), colors.HexColor('#2c3e50')),
            ('ALIGN', (0, 0), (-1, -1), 'LEFT'),
            ('VALIGN', (0, 0), (-1, -1), 'MIDDLE'),
            ('LINEBELOW', (0, -1), (-1, -1), 1, colors.grey),
            ('TOPPADDING', (0, 0), (-1, -1), 8),
            ('BOTTOMPADDING', (0, 0), (-1, -1), 8),
        ]))

        elements.append(info_table)

        elements.append(Spacer(1, 1*inch))

        # Confidentiality notice
        elements.append(Paragraph(
            "<b>CONFIDENTIAL - SECTION 45 COMPLIANT</b>",
            styles['BodyJustify']
        ))
        elements.append(Paragraph(
            "This report contains sensitive information and is intended solely for authorized
recipients. "
            "Unauthorized disclosure, distribution, or copying is prohibited.",
            styles['BodyJustify']
        ))

        return elements
```

```
    def _build_executive_summary(
        self,
        submission: Dict,
        styles: Dict
    ) -> List:
        """Build executive summary section."""
        elements = []

        elements.append(Paragraph("Executive Summary", styles['SectionHeader']))

        # Get key metrics
        credibility = submission.get('credibility', {})
        final_score = credibility.get('final_score', 0.0)
        decision = submission.get('consensus', {}).get('decision', 'review')
        coordination_flagged = submission.get('coordination', {}).get('flagged', False)

        # Summary text
        summary_text = f"""
        This forensic report presents the automated assessment of evidence submission
        <b>{submission.get('pseudonym', 'N/A')}</b>. The evidence underwent
comprehensive analysis
        through a six-layer validation framework including anonymity preservation,
credibility
        assessment, coordination detection, Byzantine consensus, and optional counter-
evidence processing.
        <br/><br/>
        <b>Key Findings:</b><br/>
        • Credibility Score: <b>{final_score:.2%}</b><br/>
        • Consensus Decision: <b>{decision.upper()}</b><br/>
        • Coordination Detected: <b>{'YES' if coordination_flagged else 'NO'}</b><br/>
        • Evidence Type: <b>{submission.get('evidence_type', 'N/A').upper()}</b>
        """

        elements.append(Paragraph(summary_text, styles['BodyJustify']))

        # Add score visualization
        score_chart_path = self._create_score_visualization(submission)
        if score_chart_path and score_chart_path.exists():
            elements.append(Spacer(1, 0.2*inch))
            img = RLImage(str(score_chart_path), width=5*inch, height=2.5*inch)
            elements.append(img)

        return elements

    def _build_chain_of_custody(
        self,
```

```
        submission: Dict,
        styles: Dict
    ) -> List:
        """Build chain of custody section."""
        elements = []

        elements.append(Paragraph("Chain of Custody", styles['SectionHeader']))

        # Get chain proof
        try:
            chain_proof = self.hash_chain.get_proof(submission.get('id'))
        except Exception:
            chain_proof = {}

        # Chain data table
        chain_data = [
            ['Field', 'Value'],
            ['Evidence Hash', submission.get('evidence_hash', 'N/A')[:32] + '...'],
            ['Chain Hash', submission.get('chain_hash', 'N/A')[:32] + '...'],
            ['Submission Time', submission.get('timestamp_submission', 'N/A')],
            ['Processing Time', f"{submission.get('processing_time_seconds', 0):.2f}s"],
            ['Chain Verified', 'YES' if chain_proof else 'PENDING']
        ]

        chain_table = Table(chain_data, colWidths=[2*inch, 4.5*inch])
        chain_table.setStyle(TableStyle([
            ('FONTNAME', (0, 0), (-1, 0), 'Helvetica-Bold'),
            ('FONTSIZE', (0, 0), (-1, -1), 9),
            ('BACKGROUND', (0, 0), (-1, 0), colors.HexColor('#ecf0f1')),
            ('TEXTCOLOR', (0, 0), (-1, 0), colors.HexColor('#2c3e50')),
            ('ALIGN', (0, 0), (-1, -1), 'LEFT'),
            ('VALIGN', (0, 0), (-1, -1), 'MIDDLE'),
            ('GRID', (0, 0), (-1, -1), 0.5, colors.grey),
            ('TOPPADDING', (0, 0), (-1, -1), 6),
            ('BOTTOMPADDING', (0, 0), (-1, -1), 6),
        ]))

        elements.append(chain_table)

        elements.append(Spacer(1, 0.1*inch))
        elements.append(Paragraph(
            "<i>Note: All evidence is cryptographically hashed and recorded in an
immutable chain "
            "to ensure integrity and prevent tampering.</i>",
            styles['BodyJustify']
        ))
```

```
    return elements

def _build_credibility_section(
    self,
    submission: Dict,
    styles: Dict
) -> List:
    """Build credibility assessment section."""
    elements = []

    elements.append(Paragraph("Credibility Assessment", styles['SectionHeader']))

    credibility = submission.get('credibility', {})

    # Scores table
    scores_data = [
        ['Metric', 'Score', 'Confidence Interval'],
        [
            'Deepfake Detection',
            f"{credibility.get('deepfake_score', 0):.2%}",
            self._format_ci(credibility.get('confidence_interval', (0, 0)))
        ],
        [
            'Cross-Modal Consistency',
            f"{credibility.get('consistency_score', 0):.2%}",
            'N/A'
        ],
        [
            'Physical Plausibility',
            f"{credibility.get('plausibility_score', 0):.2%}",
            'N/A'
        ],
        [
            '<b>Final Credibility Score</b>',
            f"<b>{credibility.get('final_score', 0):.2%}</b>",
            '<b>' + self._format_ci(credibility.get('confidence_interval', (0, 0))) + '</b>'
        ]
    ]

    scores_table = Table(scores_data, colWidths=[2.5*inch, 1.5*inch, 2.5*inch])
    scores_table.setStyle(TableStyle([
        ('FONTNAME', (0, 0), (-1, 0), 'Helvetica-Bold'),
        ('FONTSIZE', (0, 0), (-1, -1), 9),
        ('BACKGROUND', (0, 0), (-1, 0), colors.HexColor('#3498db')),
        ('TEXTCOLOR', (0, 0), (-1, 0), colors.white),
```

```
            ('ALIGN', (0, 0), (-1, -1), 'LEFT'),
            ('ALIGN', (1, 0), (-1, -1), 'CENTER'),
            ('VALIGN', (0, 0), (-1, -1), 'MIDDLE'),
            ('GRID', (0, 0), (-1, -1), 0.5, colors.grey),
            ('BACKGROUND', (0, -1), (-1, -1), colors.HexColor('#ecf0f1')),
            ('TOPPADDING', (0, 0), (-1, -1), 6),
            ('BOTTOMPADDING', (0, 0), (-1, -1), 6),
        ]))

        elements.append(scores_table)

        # Interpretation
        elements.append(Spacer(1, 0.1*inch))
        interpretation = self._interpret_credibility_score(
            credibility.get('final_score', 0)
        )
        elements.append(Paragraph(
            f"<b>Interpretation:</b> {interpretation}",
            styles['BodyJustify']
        ))

        # Uncertainty flag
        if credibility.get('entropy', 0) > 0.4:
            elements.append(Paragraph(
                "<b>&  HIGH UNCERTAINTY DETECTED:</b> This submission exhibits high
assessment "
                "uncertainty and requires human expert review.",
                styles['BodyJustify']
            ))

        return elements

    def _build_coordination_section(
        self,
        submission: Dict,
        styles: Dict
    ) -> List:
        """Build coordination detection section."""
        elements = []

        elements.append(Paragraph("Coordination Analysis", styles['SectionHeader']))

        coordination = submission.get('coordination', {})
        flagged = coordination.get('flagged', False)
        confidence = coordination.get('confidence', 0.0)
```

```
    if flagged:
        elements.append(Paragraph(
            f"<b>&  COORDINATION DETECTED</b> (Confidence: {confidence:.1%})",
            styles['BodyJustify']
        ))
        elements.append(Paragraph(
            "This submission appears to be part of a coordinated attack pattern. "
            "Multiple submissions with similar stylometric features, temporal clustering, "
            "and content overlap were identified.",
            styles['BodyJustify']
        ))

        # Community info
        community_size = coordination.get('community_size', 0)
        if community_size > 0:
            elements.append(Paragraph(
                f"<b>Community Size:</b> {community_size} submissions<br/>"
                f"<b>Average Similarity:</b> {coordination.get('avg_similarity', 0):.1%}",
                styles['BodyJustify']
            ))
    else:
        elements.append(Paragraph(
            "<b> NO COORDINATION DETECTED</b>",
            styles['BodyJustify']
        ))
        elements.append(Paragraph(
            "No evidence of coordination with other submissions was found. "
            "The submission appears to be independent.",
            styles['BodyJustify']
        ))

    return elements

def _build_consensus_section(
    self,
    submission: Dict,
    styles: Dict
) -> List:
    """Build consensus results section."""
    elements = []

    elements.append(Paragraph("Consensus Decision", styles['SectionHeader']))

    consensus = submission.get('consensus', {})
    decision = consensus.get('decision', 'review')
    agreement = consensus.get('agreement_percentage', 0)
```

```
    votes = consensus.get('votes', {})

    # Decision box
    decision_color = {
        'forward': colors.HexColor('#27ae60'),
        'reject': colors.HexColor('#e74c3c'),
        'review': colors.HexColor('#f39c12')
    }.get(decision, colors.grey)

    decision_data = [[
        f"DECISION: {decision.upper()}",
        f"Agreement: {agreement:.1%}"
    ]]

    decision_table = Table(decision_data, colWidths=[3.5*inch, 3*inch])
    decision_table.setStyle(TableStyle([
        ('FONTNAME', (0, 0), (-1, -1), 'Helvetica-Bold'),
        ('FONTSIZE', (0, 0), (-1, -1), 12),
        ('BACKGROUND', (0, 0), (-1, -1), decision_color),
        ('TEXTCOLOR', (0, 0), (-1, -1), colors.white),
        ('ALIGN', (0, 0), (-1, -1), 'CENTER'),
        ('VALIGN', (0, 0), (-1, -1), 'MIDDLE'),
        ('TOPPADDING', (0, 0), (-1, -1), 12),
        ('BOTTOMPADDING', (0, 0), (-1, -1), 12),
    ]))

    elements.append(decision_table)

    # Vote distribution
    elements.append(Spacer(1, 0.1*inch))
    elements.append(Paragraph(
        f"<b>Vote Distribution:</b> Accept: {votes.get('accept', 0)} | "
        f"Reject: {votes.get('reject', 0)}",
        styles['BodyJustify']
    ))

    # Decision explanation
    decision_text = {
        'forward': "The evidence meets credibility thresholds and consensus
requirements. "
                   "Recommended for forwarding to relevant authorities.",
        'reject': "The evidence does not meet minimum credibility requirements. "
                  "Insufficient basis for further action.",
        'review': "The evidence requires human expert review due to borderline scores, "
                  "high uncertainty, or lack of clear consensus."
    }.get(decision, "Status unknown.")
```

```
        elements.append(Paragraph(decision_text, styles['BodyJustify']))

        return elements

    def _build_counter_evidence_section(
        self,
        submission: Dict,
        styles: Dict
    ) -> List:
        """Build counter-evidence section."""
        elements = []

        elements.append(Paragraph("Counter-Evidence Analysis",
styles['SectionHeader']))

        counter_id = submission.get('counter_evidence_id')

        if not counter_id:
            elements.append(Paragraph("No counter-evidence submitted.",
styles['BodyJustify']))
            return elements

        # Load counter-evidence
        try:
            counter = self.storage.load_submission(counter_id)
        except Exception:
            counter = None

        elements.append(Paragraph(
            "<b>Counter-evidence has been submitted and processed.</b>",
            styles['BodyJustify']
        ))

        # Comparison table
        comparison_data = [
            ['Metric', 'Original', 'After Counter-Evidence'],
            [
                'Credibility Score',
                f"{submission.get('credibility', {}).get('final_score', 0):.2%}",
                f"{submission.get('posterior_score', 0):.2%}"
            ],
            [
                'Decision',
                submission.get('consensus', {}).get('decision', 'N/A').upper(),
                submission.get('new_decision', 'N/A').upper()
```

```
            ],
            [
                'Identity Verified',
                'N/A',
                'YES' if submission.get('identity_verified') else 'NO'
            ]
        ]

        comparison_table = Table(comparison_data, colWidths=[2.2*inch, 2.2*inch,
2.2*inch])
        comparison_table.setStyle(TableStyle([
            ('FONTNAME', (0, 0), (-1, 0), 'Helvetica-Bold'),
            ('FONTSIZE', (0, 0), (-1, -1), 9),
            ('BACKGROUND', (0, 0), (-1, 0), colors.HexColor('#9b59b6')),
            ('TEXTCOLOR', (0, 0), (-1, 0), colors.white),
            ('ALIGN', (0, 0), (-1, -1), 'CENTER'),
            ('VALIGN', (0, 0), (-1, -1), 'MIDDLE'),
            ('GRID', (0, 0), (-1, -1), 0.5, colors.grey),
            ('TOPPADDING', (0, 0), (-1, -1), 6),
            ('BOTTOMPADDING', (0, 0), (-1, -1), 6),
        ]))

        elements.append(comparison_table)

        # Score change
        score_delta = submission.get('score_delta', 0)
        elements.append(Spacer(1, 0.1*inch))
        elements.append(Paragraph(
            f"<b>Score Change:</b> {score_delta:+.1%} "
            f"({'increased' if score_delta > 0 else 'decreased' if score_delta < 0 else
'unchanged'})",
            styles['BodyJustify']
        ))

        return elements

    def _build_technical_details(
        self,
        submission: Dict,
        styles: Dict
    ) -> List:
        """Build technical details section."""
        elements = []

        elements.append(Paragraph("Technical Details", styles['SectionHeader']))
```

```
        # Processing timeline
        elements.append(Paragraph("<b>Processing Timeline:</b>", styles['BodyJustify']))

        timeline_data = [
            ['Stage', 'Timestamp', 'Duration'],
            [
                'Submission',
                submission.get('timestamp_submission', 'N/A')[:19],
                '-'
            ],
            [
                'Anonymization',
                submission.get('timestamp_anonymized', 'N/A')[:19],
                f"{submission.get('layer1_time', 0):.2f}s"
            ],
            [
                'Credibility Assessment',
                submission.get('timestamp_assessed', 'N/A')[:19] if
isinstance(submission.get('timestamp_assessed'), str) else 'N/A',
                f"{submission.get('credibility', {}).get('processing_time', 0):.2f}s"
            ],
            [
                'Consensus',
                submission.get('timestamp_consensus', 'N/A')[:19] if
isinstance(submission.get('timestamp_consensus'), str) else 'N/A',
                f"{submission.get('consensus_time', 0):.2f}s"
            ]
        ]

        timeline_table = Table(timeline_data, colWidths=[2.5*inch, 2.5*inch, 1.5*inch])
        timeline_table.setStyle(TableStyle([
            ('FONTNAME', (0, 0), (-1, 0), 'Helvetica-Bold'),
            ('FONTSIZE', (0, 0), (-1, -1), 8),
            ('BACKGROUND', (0, 0), (-1, 0), colors.HexColor('#ecf0f1')),
            ('ALIGN', (0, 0), (-1, -1), 'LEFT'),
            ('VALIGN', (0, 0), (-1, -1), 'MIDDLE'),
            ('GRID', (0, 0), (-1, -1), 0.5, colors.grey),
            ('TOPPADDING', (0, 0), (-1, -1), 4),
            ('BOTTOMPADDING', (0, 0), (-1, -1), 4),
        ]))

        elements.append(timeline_table)

        return elements

    def _build_legal_disclaimer(self, styles: Dict) -> List:
```

```
        """Build legal disclaimer section."""
        elements = []

        elements.append(Paragraph("Legal Disclaimer", styles['SectionHeader']))

        disclaimer_text = """
        This automated assessment is provided for informational purposes only and
should not be
        considered as legal advice or definitive proof of authenticity. The system uses pre-
trained
        machine learning models and heuristic analysis which may have limitations and
potential errors.
        <br/><br/>
        <b>Section 45 Compliance:</b> This report is generated in compliance with
Section 45 of the
        Evidence Act, which governs the admissibility of electronic evidence. However,
final
        determination of admissibility rests with the appropriate legal authorities.
        <br/><br/>
        <b>Limitations:</b><br/>
        • Automated assessments may not detect all forms of manipulation<br/>
        • Results should be validated by human experts where applicable<br/>
        • The system operates on submitted evidence without external verification<br/>
        • Coordination detection is based on patterns and may have false positives/
negatives
        <br/><br/>
        <b>Recommended Actions:</b> All evidence marked for "REVIEW" or with high
uncertainty
        should undergo manual expert examination before being used in legal
proceedings.
        """

        elements.append(Paragraph(disclaimer_text, styles['BodyJustify']))

        # Signature block
        elements.append(Spacer(1, 0.5*inch))
        elements.append(Paragraph(
            f"<i>Report generated by Corruption Reporting System v1.0.0-MVP<br/>"
            f"Generated: {datetime.utcnow().strftime('%Y-%m-%d %H:%M:%S UTC')}</i>",
            styles['BodyJustify']
        ))

        return elements

    def _create_score_visualization(self, submission: Dict) -> Optional[Path]:
        """Create credibility score visualization chart."""
```

```
        try:
            credibility = submission.get('credibility', {})

            scores = {
                'Deepfake\nDetection': credibility.get('deepfake_score', 0),
                'Cross-Modal\nConsistency': credibility.get('consistency_score', 0),
                'Physical\nPlausibility': credibility.get('plausibility_score', 0),
                'Final\nScore': credibility.get('final_score', 0)
            }

            fig, ax = plt.subplots(figsize=(8, 4))

            bars = ax.barh(list(scores.keys()), list(scores.values()),
                           color=['#3498db', '#2ecc71', '#f39c12', '#e74c3c'])

            ax.set_xlabel('Score', fontsize=10)
            ax.set_xlim(0, 1.0)
            ax.set_title('Credibility Assessment Scores', fontsize=12, fontweight='bold')
            ax.grid(axis='x', alpha=0.3)

            # Add value labels
            for i, (bar, value) in enumerate(zip(bars, scores.values())):
                ax.text(value + 0.02, bar.get_y() + bar.get_height()/2,
                        f'{value:.1%}', va='center', fontsize=9)

            plt.tight_layout()

            # Save
            chart_path = self.output_dir / f"chart_{submission.get('id', 'temp')[:8]}.png"
            plt.savefig(chart_path, dpi=150, bbox_inches='tight')
            plt.close()

            return chart_path

        except Exception as e:
            logger.warning(f"Failed to create visualization: {e}")
            return None

    def _get_classification_label(self, submission: Dict) -> str:
        """Get classification label based on decision."""
        decision = submission.get('consensus', {}).get('decision', 'review')
        return {
            'forward': 'CREDIBLE - FORWARD TO AUTHORITIES',
            'reject': 'NOT CREDIBLE - REJECTED',
            'review': 'REQUIRES HUMAN REVIEW'
        }.get(decision, 'UNKNOWN')
```

```
    def _format_ci(self, ci: tuple) -> str:
        """Format confidence interval."""
        if not ci or len(ci) != 2:
            return 'N/A'
        return f"[{ci[0]:.1%}, {ci[1]:.1%}]"

    def _interpret_credibility_score(self, score: float) -> str:
        """Interpret credibility score."""
        if score >= 0.80:
            return "HIGH credibility - Evidence appears authentic with high confidence."
        elif score >= 0.60:
            return "MODERATE credibility - Evidence appears reasonably authentic."
        elif score >= 0.40:
            return "LOW credibility - Evidence authenticity is questionable."
        else:
            return "VERY LOW credibility - Evidence likely manipulated or inauthentic."

"""
Orchestrator - Coordinates all 6 layers of the framework

Manages the complete workflow:
1. Evidence Submission: Layer 1 !’ 2 !’ 3 !’ 4
2. Counter-Evidence: Layer 5 (Bayesian aggregation)
3. Report Generation: Layer 6
"""

import asyncio
import logging
import time
from datetime import datetime
from pathlib import Path
from typing import Dict, Optional

from backend.core.layer1_anonymity import Layer1Anonymity
from backend.core.layer2_credibility import Layer2Credibility
from backend.core.layer3_coordination import Layer3Coordination
from backend.core.layer4_consensus import Layer4Consensus
from backend.core.layer5_counter_evidence import Layer5CounterEvidence
from backend.core.layer6_reporting import Layer6Reporting

from backend.services.metrics_service import MetricsService

# Initialize logger
logger = logging.getLogger(__name__)
```

```
class ProcessingStatus:
    """Processing status constants."""
    PENDING = "pending"
    PROCESSING = "processing"
    LAYER1_COMPLETE = "layer1_complete"
    LAYER2_COMPLETE = "layer2_complete"
    LAYER3_COMPLETE = "layer3_complete"
    LAYER4_COMPLETE = "layer4_complete"
    LAYER5_COMPLETE = "layer5_complete"
    COMPLETED = "completed"
    FAILED = "failed"


class Orchestrator:
    """
    Orchestrator - Coordinates all 6 layers

    Responsibilities:
    - Initialize all layers with required services
    - Process submissions through layers 1-4 sequentially
    - Handle counter-evidence processing (layer 5)
    - Generate forensic reports (layer 6)
    - Manage errors and rollback
    - Track processing metrics
    """

    def __init__(
        self,
        storage_service,
        hash_chain_service,
        crypto_service,
        metadata_service,
        validation_service,
        text_utils,
        graph_utils,
        image_utils=None,
        audio_utils=None,
        metrics_service: Optional[MetricsService] = None
    ):
        """
        Initialize orchestrator with all required services.

        Args:
            storage_service: Storage service for submissions
            hash_chain_service: Hash chain service for custody
```

```
        crypto_service: Cryptography service
        metadata_service: Metadata stripping service
        validation_service: Input validation service
        text_utils: Text utilities for stylometric analysis
        graph_utils: Graph utilities for coordination detection
        image_utils: Image utilities (optional)
        audio_utils: Audio utilities (optional)
    """
    self.storage = storage_service
    self.hash_chain = hash_chain_service
    self.crypto = crypto_service
    self.metadata = metadata_service
    self.validation = validation_service
    self.text_utils = text_utils
    self.graph_utils = graph_utils
    self.image_utils = image_utils
    self.audio_utils = audio_utils

    # Initialize all 6 layers
    logger.info("Initializing Orchestrator with 6-layer framework...")

    self.layer1 = Layer1Anonymity(
        storage_service=storage_service,
        hash_chain_service=hash_chain_service,
        crypto_service=crypto_service,
        metadata_service=metadata_service,
        metrics_service=metrics_service

    )

    self.layer2 = Layer2Credibility(
        storage_service=storage_service,
        validation_service=validation_service,
        image_utils=image_utils,
        audio_utils=audio_utils,
        metrics_service=metrics_service

    )

    self.layer3 = Layer3Coordination(
        storage_service=storage_service,
        text_utils=text_utils,
        graph_utils=graph_utils,
        metrics_service=metrics_service,
    )
```

```
        self.layer4 = Layer4Consensus(
            storage_service=storage_service,
            metrics_service=metrics_service

        )

        self.layer5 = Layer5CounterEvidence(
            storage_service=storage_service,
            metrics_service=metrics_service

        )

        self.layer6 = Layer6Reporting(
            storage_service=storage_service,
            hash_chain_service=hash_chain_service,
            metrics_service=metrics_service

        )

        logger.info("Orchestrator initialized successfully with all 6 layers")

    async def process_submission(
        self,
        submission_id: str,
        file_path: Path,
        evidence_type: str,
        text_narrative: Optional[str] = None,
        metadata: Optional[Dict] = None
    ) -> Dict:
        """
        Process evidence submission through layers 1-4.

        Workflow:
        1. Layer 1: Anonymize and secure evidence
        2. Layer 2: Assess credibility (deepfake detection)
        3. Layer 3: Detect coordination patterns
        4. Layer 4: Byzantine consensus

        Args:
            submission_id: Unique submission identifier
            file_path: Path to evidence file
            evidence_type: Type of evidence (image/video/audio/document)
            text_narrative: Optional text narrative
            metadata: Optional submission metadata

        Returns:
```

```
    dict: Complete processing results from all layers

Raises:
    Exception: If processing fails at any layer
"""
start_time = time.time()
logger.info(f"Orchestrator processing submission {submission_id}")

# Initialize result dictionary
result = {
    "id": submission_id,
    "evidence_type": evidence_type,
    "status": ProcessingStatus.PROCESSING,
    "text_narrative": text_narrative,
    "metadata": metadata or {},
    "timestamp_submission": datetime.utcnow().isoformat(),
    "processing_started": datetime.utcnow().isoformat()
}

try:
    # Update status
    self._update_status(submission_id, ProcessingStatus.PROCESSING)

    # ===== LAYER 1: ANONYMITY =====
    logger.info(f"[{submission_id}] Starting Layer 1: Anonymity")
    layer1_start = time.time()

    layer1_result = await asyncio.to_thread(
        self.layer1.process,
        submission_id=submission_id,
        file_path=file_path,
        evidence_type=evidence_type
    )

    layer1_time = time.time() - layer1_start
    result.update(layer1_result)
    result['layer1_time'] = layer1_time

    logger.info(
        f"[{submission_id}] Layer 1 complete "
        f"(pseudonym={layer1_result.get('pseudonym')}, time={layer1_time:.2f}s)"
    )

    self._update_status(submission_id, ProcessingStatus.LAYER1_COMPLETE)
    self._save_checkpoint(submission_id, result)
```

```
# ===== LAYER 2: CREDIBILITY ASSESSMENT =====
logger.info(f"[{submission_id}] Starting Layer 2: Credibility Assessment")
layer2_start = time.time()

# Use anonymized file path
anonymized_path = Path(layer1_result.get('file_path_anonymized', file_path))

layer2_result = await asyncio.to_thread(
    self.layer2.process,
    submission_id=submission_id,
    file_path=anonymized_path,
    evidence_type=evidence_type,
    text_narrative=text_narrative
)

layer2_time = time.time() - layer2_start
result['credibility'] = layer2_result
result['layer2_time'] = layer2_time

logger.info(
    f"[{submission_id}] Layer 2 complete "
    f"(score={layer2_result.get('final_score', 0):.3f}, time={layer2_time:.2f}s)"
)

self._update_status(submission_id, ProcessingStatus.LAYER2_COMPLETE)
self._save_checkpoint(submission_id, result)

# ===== LAYER 3: COORDINATION DETECTION =====
logger.info(f"[{submission_id}] Starting Layer 3: Coordination Detection")
layer3_start = time.time()

layer3_result = await asyncio.to_thread(
    self.layer3.process,
    submission_id=submission_id,
    text_narrative=text_narrative,
    timestamp=datetime.utcnow()
)

layer3_time = time.time() - layer3_start
result['coordination'] = layer3_result
result['layer3_time'] = layer3_time

logger.info(
    f"[{submission_id}] Layer 3 complete "
    f"(flagged={layer3_result.get('flagged')}, time={layer3_time:.2f}s)"
)
```

```
        self._update_status(submission_id, ProcessingStatus.LAYER3_COMPLETE)
        self._save_checkpoint(submission_id, result)

        # ===== LAYER 4: CONSENSUS =====
        logger.info(f"[{submission_id}] Starting Layer 4: Byzantine Consensus")
        layer4_start = time.time()

        layer4_result = await asyncio.to_thread(
            self.layer4.process,
            submission_id=submission_id,
            credibility_score=layer2_result.get('final_score', 0.5),
            coordination_flagged=layer3_result.get('flagged', False),
            coordination_confidence=layer3_result.get('confidence', 0.0)
        )

        layer4_time = time.time() - layer4_start
        result['consensus'] = layer4_result
        result['layer4_time'] = layer4_time

        logger.info(
            f"[{submission_id}] Layer 4 complete "
            f"(decision={layer4_result.get('decision')}, time={layer4_time:.2f}s)"
        )

        self._update_status(submission_id, ProcessingStatus.LAYER4_COMPLETE)

        # ===== FINALIZE =====
        total_time = time.time() - start_time
        result['processing_time_seconds'] = total_time
        result['status'] = ProcessingStatus.COMPLETED
        result['timestamp_completed'] = datetime.utcnow().isoformat()

        # Save final result
        self.storage.save_submission(submission_id, result)
        self._update_status(submission_id, ProcessingStatus.COMPLETED)

        logger.info(
            f"[{submission_id}] Processing completed successfully "
            f"(total_time={total_time:.2f}s)"
        )

        return result

    except Exception as e:
        logger.error(
```

```
            f"[{submission_id}] Processing failed: {e}",
            exc_info=True
        )

        # Mark as failed
        result['status'] = ProcessingStatus.FAILED
        result['error'] = str(e)
        result['timestamp_failed'] = datetime.utcnow().isoformat()

        self.storage.save_submission(submission_id, result)
        self._update_status(submission_id, ProcessingStatus.FAILED)

        raise

async def process_counter_evidence(
    self,
    original_submission_id: str,
    counter_evidence_id: str,
    counter_credibility_score: float,
    identity_verified: bool = False
) -> Dict:
    """
    Process counter-evidence through Layer 5.

    Args:
        original_submission_id: Original submission ID
        counter_evidence_id: Counter-evidence submission ID
        counter_credibility_score: Credibility score of counter-evidence
        identity_verified: Whether defense verified identity

    Returns:
        dict: Layer 5 results with updated scores and decision

    Raises:
        Exception: If processing fails
    """
    logger.info(
        f"Orchestrator processing counter-evidence: "
        f"{counter_evidence_id} for {original_submission_id}"
    )

    try:
        # ===== LAYER 5: COUNTER-EVIDENCE =====
        layer5_start = time.time()

        layer5_result = self.layer5.process(
```

```
                original_submission_id=original_submission_id,
                counter_evidence_id=counter_evidence_id,
                counter_credibility_score=counter_credibility_score,
                identity_verified=identity_verified
            )

            layer5_time = time.time() - layer5_start
            layer5_result['layer5_time'] = layer5_time

            logger.info(
                f"Layer 5 complete: {original_submission_id} "
                f"(posterior={layer5_result.get('posterior_score'):.3f}, "
                f"decision_changed={layer5_result.get('decision_changed')})"
            )

            # Update original submission with Layer 5 results
            original = self.storage.load_submission(original_submission_id)
            if original:
                original.update({
                    'counter_evidence_id': counter_evidence_id,
                    'posterior_score': layer5_result.get('posterior_score'),
                    'score_delta': layer5_result.get('score_delta'),
                    'new_decision': layer5_result.get('new_decision'),
                    'decision_changed': layer5_result.get('decision_changed'),
                    'identity_verified': identity_verified,
                    'layer5_result': layer5_result,
                    'timestamp_counter_evidence': datetime.utcnow().isoformat()
                })

                self.storage.save_submission(original_submission_id, original)
                self._update_status(original_submission_id,
ProcessingStatus.LAYER5_COMPLETE)

            return layer5_result

        except Exception as e:
            logger.error(f"Counter-evidence processing failed: {e}", exc_info=True)
            raise

    async def generate_report(
        self,
        submission_id: str,
        include_technical_details: bool = True
    ) -> Path:
        """
        Generate forensic report through Layer 6.
```

```
    Args:
        submission_id: Submission identifier
        include_technical_details: Include technical analysis details

    Returns:
        Path: Path to generated PDF report

    Raises:
        Exception: If report generation fails
    """
    logger.info(f"Orchestrator generating report for {submission_id}")

    try:
        # ===== LAYER 6: REPORT GENERATION =====
        layer6_start = time.time()

        report_path = self.layer6.generate_report(
            submission_id=submission_id,
            include_technical_details=include_technical_details
        )

        layer6_time = time.time() - layer6_start

        logger.info(
            f"Layer 6 complete: {submission_id} "
            f"(report={report_path.name}, time={layer6_time:.2f}s)"
        )

        # Update submission with report info
        submission = self.storage.load_submission(submission_id)
        if submission:
            submission['report_path'] = str(report_path)
            submission['report_generated'] = datetime.utcnow().isoformat()
            submission['layer6_time'] = layer6_time
            self.storage.save_submission(submission_id, submission)

        return report_path

    except Exception as e:
        logger.error(f"Report generation failed: {e}", exc_info=True)
        raise

def get_submission_status(self, submission_id: str) -> Dict:
    """
    Get current processing status of submission.
```

```
    Args:
        submission_id: Submission identifier

    Returns:
        dict: Status information
    """
    try:
        submission = self.storage.load_submission(submission_id)

        if not submission:
            return {
                "id": submission_id,
                "status": "not_found",
                "error": "Submission not found"
            }

        # Extract key status info
        status_info = {
            "id": submission_id,
            "status": submission.get('status', 'unknown'),
            "pseudonym": submission.get('pseudonym'),
            "evidence_type": submission.get('evidence_type'),
            "timestamp_submission": submission.get('timestamp_submission'),
            "processing_time": submission.get('processing_time_seconds'),
            "layers_completed": self._get_completed_layers(submission),
            "credibility_score": submission.get('credibility', {}).get('final_score'),
            "decision": submission.get('consensus', {}).get('decision'),
            "coordination_flagged": submission.get('coordination', {}).get('flagged'),
            "report_available": 'report_path' in submission
        }

        return status_info

    except Exception as e:
        logger.error(f"Failed to get submission status: {e}")
        return {
            "id": submission_id,
            "status": "error",
            "error": str(e)
        }

def get_system_statistics(self) -> Dict:
    """
    Get system-wide statistics.
```

```
        Returns:
            dict: System statistics
        """
        try:
            all_submissions = self.storage.get_all_submissions()

            # Calculate statistics
            total_submissions = len(all_submissions)
            completed = sum(1 for s in all_submissions if s.get('status') ==
ProcessingStatus.COMPLETED)
            failed = sum(1 for s in all_submissions if s.get('status') ==
ProcessingStatus.FAILED)
            processing = total_submissions - completed - failed

            # Credibility statistics
            scores = [
                s.get('credibility', {}).get('final_score', 0)
                for s in all_submissions
                if s.get('credibility')
            ]

            # Decision distribution
            decisions = {}
            for s in all_submissions:
                decision = s.get('consensus', {}).get('decision')
                if decision:
                    decisions[decision] = decisions.get(decision, 0) + 1

            # Coordination detection
            coordinated = sum(
                1 for s in all_submissions
                if s.get('coordination', {}).get('flagged', False)
            )

            # Processing times
            processing_times = [
                s.get('processing_time_seconds', 0)
                for s in all_submissions
                if s.get('processing_time_seconds')
            ]

            import numpy as np

            stats = {
                "total_submissions": total_submissions,
                "completed": completed,
```

```
                "failed": failed,
                "processing": processing,
                "completion_rate": completed / total_submissions if total_submissions > 0
else 0,
                "credibility_scores": {
                    "mean": float(np.mean(scores)) if scores else 0,
                    "median": float(np.median(scores)) if scores else 0,
                    "std": float(np.std(scores)) if scores else 0,
                    "min": float(np.min(scores)) if scores else 0,
                    "max": float(np.max(scores)) if scores else 0
                },
                "decisions": decisions,
                "coordination_detected": coordinated,
                "coordination_rate": coordinated / total_submissions if total_submissions > 0
else 0,
                "processing_times": {
                    "mean": float(np.mean(processing_times)) if processing_times else 0,
                    "median": float(np.median(processing_times)) if processing_times else 0,
                    "min": float(np.min(processing_times)) if processing_times else 0,
                    "max": float(np.max(processing_times)) if processing_times else 0
                },
                "validator_stats": self.layer4.get_validator_statistics(),
                "timestamp": datetime.utcnow().isoformat()
            }

            return stats

        except Exception as e:
            logger.error(f"Failed to get system statistics: {e}")
            return {
                "error": str(e),
                "timestamp": datetime.utcnow().isoformat()
            }

    def _update_status(self, submission_id: str, status: str) -> None:
        """
        Update submission status.

        Args:
            submission_id: Submission identifier
            status: New status
        """
        try:
            self.storage.update_submission_status(submission_id, status)
        except Exception as e:
            logger.warning(f"Failed to update status for {submission_id}: {e}")
```

```
def _save_checkpoint(self, submission_id: str, data: Dict) -> None:
    """
    Save processing checkpoint.

    Args:
        submission_id: Submission identifier
        data: Current processing data
    """
    try:
        self.storage.save_submission(submission_id, data)
    except Exception as e:
        logger.warning(f"Failed to save checkpoint for {submission_id}: {e}")

def _get_completed_layers(self, submission: Dict) -> list:
    """
    Get list of completed layers.

    Args:
        submission: Submission data

    Returns:
        list: Completed layer names
    """
    completed = []

    if submission.get('pseudonym'):
        completed.append('layer1_anonymity')

    if submission.get('credibility'):
        completed.append('layer2_credibility')

    if submission.get('coordination'):
        completed.append('layer3_coordination')

    if submission.get('consensus'):
        completed.append('layer4_consensus')

    if submission.get('layer5_result'):
        completed.append('layer5_counter_evidence')

    if submission.get('report_path'):
        completed.append('layer6_reporting')

    return completed
```

```
async def retry_failed_submission(
    self,
    submission_id: str,
    retry_from_layer: Optional[int] = None
) -> Dict:
    """
    Retry a failed submission.

    Args:
        submission_id: Submission identifier
        retry_from_layer: Layer to retry from (1-4), or None for full retry

    Returns:
        dict: Processing results

    Raises:
        Exception: If retry fails
    """
    logger.info(f"Retrying submission {submission_id} from layer {retry_from_layer}")

    try:
        # Load failed submission
        submission = self.storage.load_submission(submission_id)
        if not submission:
            raise ValueError(f"Submission {submission_id} not found")

        # Determine what to retry
        if retry_from_layer is None or retry_from_layer == 1:
            # Full retry - extract original file path
            file_path = Path(submission.get('file_path', ''))
            evidence_type = submission.get('evidence_type', 'image')
            text_narrative = submission.get('text_narrative')

            return await self.process_submission(
                submission_id=submission_id,
                file_path=file_path,
                evidence_type=evidence_type,
                text_narrative=text_narrative
            )

        else:
            # Partial retry - not implemented for MVP
            logger.warning("Partial retry not supported in MVP")
            raise NotImplementedError("Partial retry not supported in MVP")

    except Exception as e:
```

```
        logger.error(f"Retry failed for {submission_id}: {e}", exc_info=True)
        raise

def cleanup_old_submissions(self, days: int = 90) -> int:
    """
    Cleanup submissions older than specified days.

    Args:
        days: Age threshold in days

    Returns:
        int: Number of submissions cleaned up
    """
    logger.info(f"Cleaning up submissions older than {days} days")

    try:
        from datetime import timedelta

        cutoff_date = datetime.utcnow() - timedelta(days=days)
        all_submissions = self.storage.get_all_submissions()

        cleaned = 0
        for submission in all_submissions:
            timestamp_str = submission.get('timestamp_submission')
            if timestamp_str:
                try:
                    timestamp = datetime.fromisoformat(timestamp_str.replace('Z', ''))
                    if timestamp < cutoff_date:
                        # Archive and delete
                        submission_id = submission.get('id')
                        self.storage.archive_submission(submission_id)
                        cleaned += 1
                except Exception as e:
                    logger.warning(f"Failed to cleanup submission: {e}")

        logger.info(f"Cleaned up {cleaned} submissions")
        return cleaned

    except Exception as e:
        logger.error(f"Cleanup failed: {e}")
        return 0

def health_check(self) -> Dict:
    """
    Perform health check on all layers.
```

```
Returns:
    dict: Health status of each layer
"""
health = {
    "orchestrator": "healthy",
    "timestamp": datetime.utcnow().isoformat()
}

try:
    # Check Layer 1
    health['layer1_anonymity'] = "healthy" if self.layer1 else "unavailable"

    # Check Layer 2
    health['layer2_credibility'] = "healthy" if self.layer2 else "unavailable"

    # Check Layer 3
    health['layer3_coordination'] = "healthy" if self.layer3 else "unavailable"

    # Check Layer 4
    health['layer4_consensus'] = "healthy" if self.layer4 else "unavailable"

    # Check Layer 5
    health['layer5_counter_evidence'] = "healthy" if self.layer5 else "unavailable"

    # Check Layer 6
    health['layer6_reporting'] = "healthy" if self.layer6 else "unavailable"

    # Check storage
    try:
        self.storage.health_check()
        health['storage'] = "healthy"
    except Exception:
        health['storage'] = "unhealthy"

    # Check hash chain
    try:
        self.hash_chain.verify_chain()
        health['hash_chain'] = "healthy"
    except Exception:
        health['hash_chain'] = "unhealthy"

    # Overall status
    unhealthy = [k for k, v in health.items() if v == "unhealthy"]
    if unhealthy:
        health['overall'] = "degraded"
    else:
```

```
            health['overall'] = "healthy"

        except Exception as e:
            logger.error(f"Health check failed: {e}")
            health['overall'] = "unhealthy"
            health['error'] = str(e)

        return health

"""
Core Module - Business Logic Layers

This module contains the 6-layer framework implementation:
- Layer 1: Anonymous Submission Gateway
- Layer 2: Credibility Assessment Engine
- Layer 3: Coordination Detection System
- Layer 4: Byzantine Consensus Simulator
- Layer 5: Counter-Evidence Processor
- Layer 6: Forensic Report Generator

Plus the Orchestrator that coordinates all layers.
"""

from backend.core.layer1_anonymity import Layer1Anonymity
from backend.core.layer2_credibility import Layer2Credibility
from backend.core.layer3_coordination import Layer3Coordination
from backend.core.layer4_consensus import Layer4Consensus
from backend.core.layer5_counter_evidence import Layer5CounterEvidence
from backend.core.layer6_reporting import Layer6Reporting
from backend.core.orchestrator import Orchestrator
from backend.services.metrics_service import MetricsService

__all__ = [
    'Layer1Anonymity',
    'Layer2Credibility',
    'Layer3Coordination',
    'Layer4Consensus',
    'Layer5CounterEvidence',
    'Layer6Reporting',
    'Orchestrator'
]

__version__ = '1.0.0-MVP'
```