Curation Workflow Implementation - Deep Dive

Architecture Overview

The curation workflow is the heart of the platform's quality assurance system, implementing a three-stage pipeline that combines AI automation, human expertise, and community feedback to maintain consistently high-quality events.

mermaid

```
flowchart TD
    A[Event Submitted] --> B[AI Pre-Screening]
    B --> C{AI Decision}
    C -->|Pass| D[Human Review Queue]
    C -->|Fail| E[Auto-Reject with Feedback]
    C -->|Uncertain| F[Priority Human Review]

    D --> G[Curator Assignment]
    G --> H[Human Review]
    H --> I{Curator Decision}
    I -->|Approve| J[Published]
    I -->|Reject| K[Rejected with Feedback]
    I -->|Needs Revision| L[Back to Creator]

    J --> M[Community Oversight]
    M --> N[Post-Event Feedback]
    N --> O[Quality Score Update]
    O --> P[Creator Reputation Update]

    L --> Q[Creator Revisions]
    Q --> R[Re-enter Pipeline]

    style B fill:#e1f5fe
    style H fill:#f3e5f5
    style M fill:#e8f5e8
```

---

1. AI Pre-Screening System

Machine Learning Pipeline Architecture

python

```python
# AI Pre-Screening Service Implementation
import asyncio
import logging
from typing import Dict, List, Optional, Tuple
from dataclasses import dataclass
from enum import Enum
import torch
import transformers
from PIL import Image
import cv2
import numpy as np

@dataclass
class AIAssessmentResult:
    overall_score: float
    component_scores: Dict[str, float]
    confidence: float
    flags: List[str]
    recommendations: List[str]
    processing_time_ms: int

class AIScreeningStage(Enum):
    COMPLETENESS = "completeness"
    CONTENT_QUALITY = "content_quality"
    IMAGE_QUALITY = "image_quality"
    SPAM_DETECTION = "spam_detection"
    DUPLICATE_DETECTION = "duplicate_detection"
    SAFETY_SCREENING = "safety_screening"

class EventAIScreeningService:
    def __init__(self):
        self.models = {
            'content_quality': self._load_content_model(),
            'image_quality': self._load_image_model(),
            'spam_detector': self._load_spam_model(),
            'safety_classifier': self._load_safety_model(),
            'duplicate_detector': self._load_duplicate_model()
        }
        self.quality_thresholds = {
            'auto_approve': 0.85,
            'human_review': 0.60,
            'auto_reject': 0.30
        }
```

```python
async def screen_event(self, event_data: Dict) -> AIAssessmentResult:
    start_time = asyncio.get_event_loop().time()

    # Parallel processing of different assessment components
    tasks = [
        self._assess_completeness(event_data),
        self._assess_content_quality(event_data),
        self._assess_image_quality(event_data),
        self._detect_spam(event_data),
        self._detect_duplicates(event_data),
        self._screen_safety(event_data)
    ]

    results = await asyncio.gather(*tasks, return_exceptions=True)

    # Combine results and calculate overall score
    component_scores = {}
    flags = []
    recommendations = []

    for i, stage in enumerate(AIScreeningStage):
        if isinstance(results[i], Exception):
            logging.error(f"AI screening failed for {stage.value}: {results[i]}")
            component_scores[stage.value] = 0.5  # Default score on failure
            flags.append(f"ai_processing_error_{stage.value}")
        else:
            score, stage_flags, stage_recommendations = results[i]
            component_scores[stage.value] = score
            flags.extend(stage_flags)
            recommendations.extend(stage_recommendations)

    overall_score = self._calculate_weighted_score(component_scores)
    confidence = self._calculate_confidence(component_scores, flags)

    processing_time = int((asyncio.get_event_loop().time() - start_time) * 1000)

    return AIAssessmentResult(
        overall_score=overall_score,
        component_scores=component_scores,
        confidence=confidence,
        flags=flags,
        recommendations=recommendations,
        processing_time_ms=processing_time
```

```
    )

async def _assess_completeness(self, event_data: Dict) -> Tuple[float, List[str], List[str]]:
    """Assess how complete the event information is"""
    score = 0.0
    flags = []
    recommendations = []

    # Required fields check
    required_fields = ['title', 'description', 'start_time', 'end_time', 'location']
    missing_fields = [field for field in required_fields if not event_data.get(field)]

    if missing_fields:
        flags.extend([f"missing_{field}" for field in missing_fields])
        recommendations.append(f"Please provide: {', '.join(missing_fields)}")
        return 0.0, flags, recommendations

    # Quality scoring based on information richness
    title_score = min(len(event_data['title']) / 50, 1.0)  # Ideal title length ~50 chars
    desc_score = min(len(event_data['description']) / 300, 1.0)  # Ideal desc length ~300 chars

    # Bonus points for optional fields
    optional_bonuses = {
        'cover_image': 0.2,
        'gallery_images': 0.1,
        'ticket_url': 0.1,
        'venue_details': 0.1,
        'tags': 0.1
    }

    bonus_score = sum(
        bonus for field, bonus in optional_bonuses.items()
        if event_data.get(field)
    )

    score = (title_score * 0.3 + desc_score * 0.5 + bonus_score * 0.2)

    # Recommendations for improvement
    if title_score < 0.7:
        recommendations.append("Consider expanding your event title to be more descriptive")
    if desc_score < 0.7:
        recommendations.append("Add more details to your event description")
    if not event_data.get('cover_image'):
        recommendations.append("Add a compelling cover image to attract more attendees")
```

```
    return min(score, 1.0), flags, recommendations

async def _assess_content_quality(self, event_data: Dict) -> Tuple[float, List[str], List[str]]:
    """Use NLP models to assess content quality"""
    flags = []
    recommendations = []

    title = event_data.get('title', '')
    description = event_data.get('description', '')
    combined_text = f"{title}. {description}"

    # Language quality assessment using fine-tuned BERT
    try:
        inputs = self.models['content_quality']['tokenizer'](
            combined_text,
            return_tensors="pt",
            max_length=512,
            truncation=True,
            padding=True
        )

        with torch.no_grad():
            outputs = self.models['content_quality']['model'](**inputs)
            quality_score = torch.softmax(outputs.logits, dim=-1)[0][1].item()  # Probability of "high quality"

        # Grammar and readability checks
        grammar_score = await self._check_grammar(combined_text)
        readability_score = self._calculate_readability(combined_text)

        # Combined content quality score
        content_score = (quality_score * 0.5 + grammar_score * 0.3 + readability_score * 0.2)

        # Generate flags and recommendations
        if grammar_score < 0.6:
            flags.append("grammar_issues")
            recommendations.append("Consider reviewing for grammar and spelling errors")

        if readability_score < 0.5:
            flags.append("readability_low")
            recommendations.append("Simplify language to make your event more accessible")

        if quality_score < 0.4:
            flags.append("content_quality_low")
```

```
            recommendations.append("Add more specific details about what attendees can expect")

        return content_score, flags, recommendations

    except Exception as e:
        logging.error(f"Content quality assessment failed: {e}")
        return 0.5, ["content_assessment_error"], []

async def _assess_image_quality(self, event_data: Dict) -> Tuple[float, List[str], List[str]]:
    """Assess the quality of event images using computer vision"""
    flags = []
    recommendations = []

    cover_image_url = event_data.get('cover_image')
    if not cover_image_url:
        return 0.3, ["no_cover_image"], ["Add a cover image to make your event more appealing"]

    try:
        # Download and process image
        image = await self._download_image(cover_image_url)
        if image is None:
            return 0.2, ["image_download_failed"], ["Ensure your image URL is accessible"]

        # Technical quality assessment
        technical_score = self._assess_technical_quality(image)

        # Content relevance assessment using vision transformer
        relevance_score = await self._assess_image_relevance(image, event_data)

        # Aesthetic quality assessment
        aesthetic_score = await self._assess_aesthetic_quality(image)

        # Combined image quality score
        image_score = (technical_score * 0.3 + relevance_score * 0.4 + aesthetic_score * 0.3)

        # Generate flags and recommendations
        if technical_score < 0.5:
            flags.append("low_image_quality")
            recommendations.append("Use a higher resolution image for better visual appeal")

        if relevance_score < 0.4:
            flags.append("image_not_relevant")
            recommendations.append("Choose an image that better represents your event")
```

```
            if aesthetic_score < 0.4:
                flags.append("poor_aesthetic_quality")
                recommendations.append("Consider using a more visually appealing image")

            return image_score, flags, recommendations

        except Exception as e:
            logging.error(f"Image quality assessment failed: {e}")
            return 0.4, ["image_assessment_error"], []

    async def _detect_spam(self, event_data: Dict) -> Tuple[float, List[str], List[str]]:
        """Detect spam events using multiple signals"""
        flags = []
        recommendations = []
        spam_indicators = []

        title = event_data.get('title', '').lower()
        description = event_data.get('description', '').lower()

        # Keyword-based spam detection
        spam_keywords = [
            'make money fast', 'guaranteed income', 'work from home',
            'get rich quick', 'no experience needed', 'earn $$$',
            'click here', 'limited time offer', 'act now'
        ]

        keyword_matches = sum(1 for keyword in spam_keywords if keyword in title or keyword in description)
        if keyword_matches > 0:
            spam_indicators.append(f"spam_keywords_{keyword_matches}")

        # Excessive capitalization
        if len([c for c in title if c.isupper()]) / max(len(title), 1) > 0.5:
            spam_indicators.append("excessive_caps_title")

        # Excessive punctuation
        if title.count('!') > 3 or title.count('?') > 2:
            spam_indicators.append("excessive_punctuation")

        # URL spam detection
        url_count = description.count('http://') + description.count('https://')
        if url_count > 3:
            spam_indicators.append("excessive_urls")

        # ML-based spam classification
```

```
    try:
        combined_text = f"{title}. {description}"
        spam_probability = await self._classify_spam_ml(combined_text)

        if spam_probability > 0.8:
            spam_indicators.append("ml_high_spam_probability")
        elif spam_probability > 0.6:
            spam_indicators.append("ml_moderate_spam_probability")

    except Exception as e:
        logging.error(f"ML spam detection failed: {e}")
        spam_probability = 0.5

    # Calculate non-spam score (higher is better)
    spam_score = len(spam_indicators) * 0.2 + spam_probability
    non_spam_score = max(0.0, 1.0 - spam_score)

    # Generate flags and recommendations
    if spam_indicators:
        flags.extend(spam_indicators)
        recommendations.append("Ensure your event description focuses on genuine event details")

    if spam_probability > 0.7:
        flags.append("high_spam_probability")
        recommendations.append("Revise your event description to be more specific and less promotional")

    return non_spam_score, flags, recommendations

def _calculate_weighted_score(self, component_scores: Dict[str, float]) -> float:
    """Calculate weighted overall score from component scores"""
    weights = {
        'completeness': 0.25,
        'content_quality': 0.25,
        'image_quality': 0.15,
        'spam_detection': 0.20,
        'duplicate_detection': 0.10,
        'safety_screening': 0.05
    }

    return sum(
        component_scores.get(component, 0.5) * weight
        for component, weight in weights.items()
    )
```

```python
    def _calculate_confidence(self, scores: Dict[str, float], flags: List[str]) -> float:
        """Calculate confidence in the AI assessment"""
        # Higher confidence when scores are more extreme (very high or very low)
        score_variance = np.var(list(scores.values()))
        confidence_from_variance = min(score_variance * 2, 1.0)

        # Lower confidence when there are many flags
        flag_penalty = min(len(flags) * 0.1, 0.5)

        return max(0.1, confidence_from_variance - flag_penalty)
```

AI Model Training Pipeline

python

```python
# Training Pipeline for Content Quality Model
class ContentQualityTrainer:
    def __init__(self):
        self.model_name = "distilbert-base-uncased"
        self.tokenizer = transformers.AutoTokenizer.from_pretrained(self.model_name)
        self.model = transformers.AutoModelForSequenceClassification.from_pretrained(
            self.model_name,
            num_labels=3  # Low, Medium, High quality
        )

    def prepare_training_data(self):
        """Prepare training dataset from curator decisions"""
        query = """
        SELECT
            e.title,
            e.description,
            cr.quality_score,
            cr.decision,
            array_agg(cr2.quality_score) as all_scores
        FROM events e
        JOIN curation_workflows cw ON e.id = cw.event_id
        JOIN curation_reviews cr ON cw.id = cr.workflow_id
        LEFT JOIN curation_reviews cr2 ON cw.id = cr2.workflow_id
        WHERE cr.reviewer_type = 'human_curator'
          AND cr.created_at > NOW() - INTERVAL '6 months'
          AND cr.quality_score IS NOT NULL
        GROUP BY e.id, cr.id
        HAVING COUNT(cr2.id) >= 2  -- At least 2 reviews for consistency
        """

        # Fetch data and create labels
        training_data = []
        for row in self.db.execute(query):
            text = f"{row['title']}. {row['description']}"

            # Calculate consensus score
            scores = [s for s in row['all_scores'] if s is not None]
            avg_score = sum(scores) / len(scores)

            # Convert to classification labels
            if avg_score >= 8:
                label = 2  # High quality
            elif avg_score >= 6:
```

```
            label = 1  # Medium quality
        else:
            label = 0  # Low quality

        training_data.append({
            'text': text,
            'label': label,
            'score': avg_score
        })

    return training_data

def train_model(self, training_data, validation_split=0.2):
    """Train the content quality model"""
    # Split data
    split_idx = int(len(training_data) * (1 - validation_split))
    train_data = training_data[:split_idx]
    val_data = training_data[split_idx:]

    # Create datasets
    train_dataset = ContentQualityDataset(train_data, self.tokenizer)
    val_dataset = ContentQualityDataset(val_data, self.tokenizer)

    # Training arguments
    training_args = transformers.TrainingArguments(
        output_dir='./content_quality_model',
        num_train_epochs=3,
        per_device_train_batch_size=16,
        per_device_eval_batch_size=64,
        warmup_steps=500,
        weight_decay=0.01,
        logging_dir='./logs',
        logging_steps=100,
        evaluation_strategy="epoch",
        save_strategy="epoch",
        load_best_model_at_end=True,
        metric_for_best_model="eval_accuracy",
    )

    # Trainer
    trainer = transformers.Trainer(
        model=self.model,
        args=training_args,
        train_dataset=train_dataset,
```

```
        eval_dataset=val_dataset,
        compute_metrics=self.compute_metrics,
    )

    # Train
    trainer.train()

    # Save model
    trainer.save_model('./content_quality_model_final')
    self.tokenizer.save_pretrained('./content_quality_model_final')

def compute_metrics(self, eval_pred):
    """Calculate evaluation metrics"""
    predictions, labels = eval_pred
    predictions = np.argmax(predictions, axis=1)

    accuracy = (predictions == labels).mean()

    # Calculate per-class metrics
    from sklearn.metrics import classification_report, confusion_matrix

    report = classification_report(labels, predictions, output_dict=True)

    return {
        'accuracy': accuracy,
        'f1_macro': report['macro avg']['f1-score'],
        'precision_macro': report['macro avg']['precision'],
        'recall_macro': report['macro avg']['recall']
    }
```

---

2. Human Review System

Curator Dashboard Implementation

typescript

```typescript
// Curator Dashboard Service
interface CuratorDashboard {
  getReviewQueue(curatorId: string, filters: QueueFilters): Promise<ReviewQueueItem[]>;
  claimReviewTask(curatorId: string, workflowId: string): Promise<CurationWorkflow>;
  submitReview(review: CurationReviewSubmission): Promise<CurationWorkflow>;
  getCuratorStats(curatorId: string, timeframe: TimeFrame): Promise<CuratorStats>;
}

interface ReviewQueueItem {
  workflowId: string;
  eventId: string;
  priority: Priority;
  aiAssessment: AIAssessmentResult;
  estimatedReviewTime: number;
  submittedAt: Date;
  creatorReputation: number;
  eventPreview: EventPreview;
}

interface CurationReviewSubmission {
  workflowId: string;
  curatorId: string;
  decision: 'approve' | 'reject' | 'needs_revision';
  qualityScore: number;
  detailedScores: {
    contentQuality: number;
    accuracy: number;
    relevance: number;
    presentation: number;
  };
  notes: string;
  flags: string[];
  improvementSuggestions: string[];
  reviewDurationSeconds: number;
}

class CuratorService {
  constructor(
    private db: DatabaseService,
    private eventService: EventService,
    private notificationService: NotificationService,
    private analyticsService: AnalyticsService
  ) {}
```

```
async getReviewQueue(curatorId: string, filters: QueueFilters): Promise<ReviewQueueItem[]> {
 // Smart queue prioritization algorithm
 const baseQuery = `
  WITH curator_specialties AS (
   SELECT category_id, AVG(quality_score) as avg_score, COUNT(*) as review_count
   FROM curation_reviews cr
   JOIN curation_workflows cw ON cr.workflow_id = cw.id
   JOIN events e ON cw.event_id = e.id
   WHERE cr.reviewer_id = $1
    AND cr.created_at > NOW() - INTERVAL '6 months'
   GROUP BY category_id
  ),
  priority_scores AS (
   SELECT
    cw.id as workflow_id,
    e.id as event_id,
    e.title,
    e.category_id,
    cw.ai_scores,
    cw.created_at,
    u.reputation,
    -- Priority calculation
    (CASE
     WHEN e.start_time < NOW() + INTERVAL '24 hours' THEN 1000
     WHEN e.start_time < NOW() + INTERVAL '3 days' THEN 500
     WHEN e.start_time < NOW() + INTERVAL '1 week' THEN 100
     ELSE 50
    END) +
    (CASE
     WHEN cs.avg_score IS NOT NULL THEN cs.avg_score * 10
     ELSE 50
    END) +
    (CASE
     WHEN cw.ai_confidence < 0.5 THEN 100
     WHEN cw.ai_confidence < 0.7 THEN 50
     ELSE 0
    END) as priority_score
   FROM curation_workflows cw
   JOIN events e ON cw.event_id = e.id
   JOIN users u ON e.creator_id = u.id
   LEFT JOIN curator_specialties cs ON e.category_id = cs.category_id
   WHERE cw.current_stage = 'human_review'
    AND cw.assigned_curator_id IS NULL
```

```
      AND cw.overall_status = 'pending'
    )
    SELECT *,
      EXTRACT(EPOCH FROM (NOW() - created_at)) / 3600 as hours_waiting
    FROM priority_scores
    ORDER BY priority_score DESC, created_at ASC
    LIMIT $2
  `;

  const result = await this.db.query(baseQuery, [curatorId, filters.limit || 20]);

  return result.map(row => ({
    workflowId: row.workflow_id,
    eventId: row.event_id,
    priority: this.calculatePriority(row.priority_score),
    aiAssessment: row.ai_scores,
    estimatedReviewTime: this.estimateReviewTime(row),
    submittedAt: row.created_at,
    creatorReputation: row.reputation,
    eventPreview: {
      title: row.title,
      category: row.category_id,
      hoursWaiting: row.hours_waiting
    }
  }));
}

async claimReviewTask(curatorId: string, workflowId: string): Promise<CurationWorkflow> {
  // Atomic claim operation with conflict detection
  const claimQuery = `
    UPDATE curation_workflows
    SET assigned_curator_id = $1, assigned_at = NOW()
    WHERE id = $2
      AND assigned_curator_id IS NULL
      AND current_stage = 'human_review'
    RETURNING *
  `;

  const result = await this.db.query(claimQuery, [curatorId, workflowId]);

  if (result.length === 0) {
    throw new Error('Task already claimed or not available for review');
  }
```

```
  // Log the claim for analytics
  await this.analyticsService.track({
    type: 'curation_task_claimed',
    curatorId,
    workflowId,
    timestamp: new Date()
  });

  return result[0];
}

async submitReview(review: CurationReviewSubmission): Promise<CurationWorkflow> {
  const transaction = await this.db.beginTransaction();

  try {
    // Insert detailed review record
    await transaction.query(`
      INSERT INTO curation_reviews (
        workflow_id, reviewer_id, reviewer_type, stage, decision,
        quality_score, content_quality_score, accuracy_score,
        relevance_score, presentation_score, notes, flags,
        improvement_suggestions, review_duration_seconds
      ) VALUES ($1, $2, 'human_curator', 'human_review', $3, $4, $5, $6, $7, $8, $9, $10, $11, $12)
    `, [
      review.workflowId, review.curatorId, review.decision, review.qualityScore,
      review.detailedScores.contentQuality, review.detailedScores.accuracy,
      review.detailedScores.relevance, review.detailedScores.presentation,
      review.notes, review.flags, review.improvementSuggestions,
      review.reviewDurationSeconds
    ]);

    // Update workflow status
    const nextStage = review.decision === 'approve' ? 'community_oversight' : 'completed';
    const overallStatus = this.mapDecisionToStatus(review.decision);

    const updatedWorkflow = await transaction.query(`
      UPDATE curation_workflows
      SET
        current_stage = $1,
        overall_status = $2,
        human_review_completed_at = NOW(),
        human_review_passed = $3,
        curator_quality_score = $4,
        final_quality_score = $5
```

```
        WHERE id = $6
        RETURNING *
      `, [
        nextStage, overallStatus,
        review.decision === 'approve',
        review.qualityScore,
        review.qualityScore, // Will be updated later by community feedback
        review.workflowId
      ]);

      // Update event status
      if (review.decision === 'approve') {
        await transaction.query(`
          UPDATE events
          SET status = 'approved', published_at = NOW()
          WHERE id = (SELECT event_id FROM curation_workflows WHERE id = $1)
        `, [review.workflowId]);
      } else if (review.decision === 'reject') {
        await transaction.query(`
          UPDATE events
          SET status = 'rejected'
          WHERE id = (SELECT event_id FROM curation_workflows WHERE id = $1)
        `, [review.workflowId]);
      }

      await transaction.commit();

      // Send notifications
      await this.sendReviewNotifications(review, updatedWorkflow[0]);

      // Update curator performance metrics
      await this.updateCuratorMetrics(review.curatorId, review);

      return updatedWorkflow[0];

    } catch (error) {
      await transaction.rollback();
      throw error;
    }
  }

  private async sendReviewNotifications(
    review: CurationReviewSubmission,
    workflow: CurationWorkflow
```

```
): Promise<void> {
  const event = await this.eventService.getById(workflow.eventId);

  switch (review.decision) {
    case 'approve':
      await this.notificationService.send({
        userId: event.creatorId,
        type: 'event_approved',
        title: 'Your event has been approved!',
        message: `"${event.title}" is now live and discoverable by our community.`,
        actionUrl: `/events/${event.slug}`,
        metadata: { eventId: event.id, workflowId: review.workflowId }
      });
      break;

    case 'reject':
      await this.notificationService.send({
        userId: event.creatorId,
        type: 'event_rejected',
        title: 'Event needs improvements',
        message: `"${event.title}" needs some improvements before it can be published.`,
        actionUrl: `/events/${event.id}/edit`,
        metadata: {
          eventId: event.id,
          feedback: review.notes,
          suggestions: review.improvementSuggestions
        }
      });
      break;

    case 'needs_revision':
      await this.notificationService.send({
        userId: event.creatorId,
        type: 'event_needs_revision',
        title: 'Please revise your event',
        message: `"${event.title}" is almost ready! Please make the suggested improvements.`,
        actionUrl: `/events/${event.id}/edit`,
        metadata: {
          eventId: event.id,
          feedback: review.notes,
          suggestions: review.improvementSuggestions
        }
      });
      break;
```

```
    }
  }
}
```

Curator Performance Analytics

typescript

```typescript
// Curator Analytics & Performance Tracking
interface CuratorStats {
  reviewsCompleted: number;
  averageReviewTime: number;
  qualityConsistency: number;
  specialtyAreas: CategoryExpertise[];
  performanceMetrics: {
    accuracy: number;          // Agreement with community feedback
    efficiency: number;        // Reviews per hour
    thoroughness: number;      // Detail level of reviews
    consistency: number;       // Variance in scoring
  };
  recentTrends: {
    weeklyReviews: number[];
    qualityTrends: number[];
    timeEfficiency: number[];
  };
}

interface CategoryExpertise {
  categoryId: string;
  categoryName: string;
  reviewCount: number;
  averageQuality: number;
  expertiseLevel: 'novice' | 'proficient' | 'expert';
}

class CuratorAnalyticsService {
  async getCuratorStats(curatorId: string, timeframe: TimeFrame): Promise<CuratorStats> {
    const [
      basicStats,
      qualityMetrics,
      categoryExpertise,
      performanceData
    ] = await Promise.all([
      this.getBasicStats(curatorId, timeframe),
      this.getQualityMetrics(curatorId, timeframe),
      this.getCategoryExpertise(curatorId, timeframe),
      this.getPerformanceData(curatorId, timeframe)
    ]);

    return {
      reviewsCompleted: basicStats.reviewCount,
```

```
    averageReviewTime: basicStats.avgReviewTime,
    qualityConsistency: qualityMetrics.consistency,
    specialtyAreas: categoryExpertise,
    performanceMetrics: {
      accuracy: await this.calculateAccuracy(curatorId, timeframe),
      efficiency: this.calculateEfficiency(performanceData),
      thoroughness: qualityMetrics.thoroughness,
      consistency: qualityMetrics.consistency
    },
    recentTrends: await this.getRecentTrends(curatorId)
  };
}

private async calculateAccuracy(curatorId: string, timeframe: TimeFrame): Promise<number> {
  // Compare curator decisions with post-event community feedback
  const accuracyQuery = `
    WITH curator_decisions AS (
      SELECT
        cr.workflow_id,
        cr.quality_score as curator_score,
        cr.decision as curator_decision,
        e.id as event_id
      FROM curation_reviews cr
      JOIN curation_workflows cw ON cr.workflow_id = cw.id
      JOIN events e ON cw.event_id = e.id
      WHERE cr.reviewer_id = $1
        AND cr.reviewer_type = 'human_curator'
        AND cr.created_at >= $2
        AND e.end_time < NOW() - INTERVAL '24 hours'  -- Only completed events
    ),
    community_feedback AS (
      SELECT
        e.id as event_id,
        AVG(r.rating) as avg_community_rating,
        COUNT(r.id) as review_count
      FROM events e
      JOIN reviews r ON e.id = r.event_id
      WHERE r.created_at >= e.end_time  -- Post-event reviews only
      GROUP BY e.id
      HAVING COUNT(r.id) >= 3  -- Minimum reviews for statistical significance
    )
    SELECT
      cd.curator_score,
      cf.avg_community_rating,
```

```
      ABS(cd.curator_score - cf.avg_community_rating) as score_difference,
      CASE
        WHEN ABS(cd.curator_score - cf.avg_community_rating) <= 1.0 THEN 1
        WHEN ABS(cd.curator_score - cf.avg_community_rating) <= 2.0 THEN 0.5
        ELSE 0
      END as accuracy_score
    FROM curator_decisions cd
    JOIN community_feedback cf ON cd.event_id = cf.event_id
  `;

  const results = await this.db.query(accuracyQuery, [curatorId, timeframe.start]);

  if (results.length === 0) return 0.8; // Default score for new curators

  const totalAccuracy = results.reduce((sum, row) => sum + row.accuracy_score, 0);
  return totalAccuracy / results.length;
}

private calculateEfficiency(performanceData: any[]): number {
  if (performanceData.length === 0) return 0.5;

  const avgReviewTime = performanceData.reduce((sum, data) => sum + data.reviewTime, 0) / performanceData
  const idealReviewTime = 900; // 15 minutes in seconds

  // Efficiency score: closer to ideal time = higher score
  return Math.max(0.1, Math.min(1.0, idealReviewTime / avgReviewTime));
}

async getRecentTrends(curatorId: string): Promise<any> {
  const trendsQuery = `
    SELECT
      DATE_TRUNC('week', cr.created_at) as week,
      COUNT(*) as review_count,
      AVG(cr.quality_score) as avg_quality,
      AVG(cr.review_duration_seconds) as avg_duration
    FROM curation_reviews cr
    WHERE cr.reviewer_id = $1
      AND cr.created_at >= NOW() - INTERVAL '12 weeks'
      AND cr.reviewer_type = 'human_curator'
    GROUP BY DATE_TRUNC('week', cr.created_at)
    ORDER BY week ASC
  `;

  const trends = await this.db.query(trendsQuery, [curatorId]);
```

```
    return {
      weeklyReviews: trends.map(t => t.review_count),
      qualityTrends: trends.map(t => parseFloat(t.avg_quality)),
      timeEfficiency: trends.map(t => idealReviewTime / t.avg_duration)
    };
  }
}
```

---

3. Community Oversight System

Post-Event Feedback Collection

typescript

```typescript
// Community Review System
interface CommunityReview {
  id: string;
  eventId: string;
  userId: string;
  rating: number;
  qualityAspects: {
    accurateDescription: number;
    organization: number;
    value: number;
    venue: number;
  };
  attendanceVerified: boolean;
  feedback: string;
  helpful: boolean;
  flags: CommunityFlag[];
  createdAt: Date;
}

interface CommunityFlag {
  id: string;
  eventId: string;
  userId: string;
  flagType: FlagType;
  reason: string;
  details: string;
  status: 'pending' | 'resolved' | 'dismissed';
  priority: 'low' | 'medium' | 'high' | 'urgent';
  createdAt: Date;
}

enum FlagType {
  MISLEADING_DESCRIPTION = 'misleading_description',
  POOR_ORGANIZATION = 'poor_organization',
  SAFETY_CONCERN = 'safety_concern',
  SPAM_EVENT = 'spam_event',
  CANCELLED_NO_NOTICE = 'cancelled_no_notice',
  INAPPROPRIATE_CONTENT = 'inappropriate_content',
  VENUE_ISSUES = 'venue_issues',
  PRICING_MISLEADING = 'pricing_misleading'
}

class CommunityOversightService {
```

```
constructor(
  private db: DatabaseService,
  private reputationService: ReputationService,
  private notificationService: NotificationService,
  private mlService: MachineLearningService
) {}

async submitCommunityReview(review: CommunityReview): Promise<void> {
  // Verify user actually attended the event
  const attendance = await this.verifyAttendance(review.userId, review.eventId);
  if (!attendance) {
    throw new Error('Only verified attendees can submit reviews');
  }

  // Prevent duplicate reviews
  const existingReview = await this.db.query(
    'SELECT id FROM community_reviews WHERE user_id = $1 AND event_id = $2',
    [review.userId, review.eventId]
  );

  if (existingReview.length > 0) {
    throw new Error('User has already reviewed this event');
  }

  // Store the review
  await this.db.query(`
    INSERT INTO community_reviews (
      event_id, user_id, rating, accurate_description_score,
      organization_score, value_score, venue_score,
      attendance_verified, feedback, helpful, created_at
    ) VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9, $10, NOW())
  `, [
    review.eventId, review.userId, review.rating,
    review.qualityAspects.accurateDescription,
    review.qualityAspects.organization,
    review.qualityAspects.value,
    review.qualityAspects.venue,
    review.attendanceVerified,
    review.feedback,
    review.helpful
  ]);

  // Update event's community rating
  await this.updateEventCommunityRating(review.eventId);
```

```
    // Update creator reputation
    await this.reputationService.updateCreatorReputation(review.eventId, review.rating);

    // Check for concerning patterns
    await this.checkForQualityIssues(review.eventId, review);
  }

  async flagEvent(flag: CommunityFlag): Promise<void> {
    // Calculate user's flag credibility based on history
    const userCredibility = await this.reputationService.getFlagCredibility(flag.userId);

    // Assign priority based on flag type and user credibility
    const priority = this.calculateFlagPriority(flag.flagType, userCredibility);

    // Store the flag
    await this.db.query(`
      INSERT INTO community_flags (
        event_id, user_id, flag_type, reason, details,
        priority, status, created_at
      ) VALUES ($1, $2, $3, $4, $5, $6, 'pending', NOW())
    `, [
      flag.eventId, flag.userId, flag.flagType,
      flag.reason, flag.details, priority
    ]);

    // Auto-escalate high-priority flags
    if (priority === 'urgent' || priority === 'high') {
      await this.escalateFlag(flag);
    }

    // Check for flag clustering (multiple similar flags)
    await this.checkFlagClustering(flag.eventId, flag.flagType);
  }

  private async updateEventCommunityRating(eventId: string): Promise<void> {
    const ratingQuery = `
      WITH weighted_ratings AS (
        SELECT
          cr.rating,
          cr.accurate_description_score,
          cr.organization_score,
          cr.value_score,
          cr.venue_score,
```

```
      ur.community_score / 100.0 as user_weight  -- User reputation as weight
    FROM community_reviews cr
    JOIN user_reputation ur ON cr.user_id = ur.user_id
    WHERE cr.event_id = $1
  ),
  calculated_scores AS (
    SELECT
      SUM(rating * user_weight) / SUM(user_weight) as weighted_avg_rating,
      SUM(accurate_description_score * user_weight) / SUM(user_weight) as accuracy_score,
      SUM(organization_score * user_weight) / SUM(user_weight) as org_score,
      SUM(value_score * user_weight) / SUM(user_weight) as value_score,
      SUM(venue_score * user_weight) / SUM(user_weight) as venue_score,
      COUNT(*) as review_count
    FROM weighted_ratings
  )
  UPDATE curation_workflows
  SET
    community_average_rating = cs.weighted_avg_rating,
    community_reviews_count = cs.review_count,
    final_quality_score = (
      COALESCE(curator_quality_score, 5.0) * 0.6 +
      cs.weighted_avg_rating * 0.4
    )
  FROM calculated_scores cs
  WHERE event_id = $1
 `;

 await this.db.query(ratingQuery, [eventId]);
}

private async checkForQualityIssues(eventId: string, review: CommunityReview): Promise<void> {
 // Get recent reviews for this event
 const recentReviews = await this.db.query(`
   SELECT rating, accurate_description_score, organization_score
   FROM community_reviews
   WHERE event_id = $1
     AND created_at >= NOW() - INTERVAL '7 days'
   ORDER BY created_at DESC
   LIMIT 10
 `, [eventId]);

 if (recentReviews.length < 3) return; // Need minimum reviews

 const avgRating = recentReviews.reduce((sum, r) => sum + r.rating, 0) / recentReviews.length;
```

```
    const avgAccuracy = recentReviews.reduce((sum, r) => sum + r.accurate_description_score, 0) / recentReview

    // Trigger alerts for quality issues
    if (avgRating < 2.5) {
      await this.createQualityAlert(eventId, 'low_overall_rating', {
        averageRating: avgRating,
        reviewCount: recentReviews.length
      });
    }

    if (avgAccuracy < 2.0) {
      await this.createQualityAlert(eventId, 'misleading_description', {
        averageAccuracy: avgAccuracy,
        reviewCount: recentReviews.length
      });
    }

    // Check creator's recent event quality
    await this.checkCreatorQualityPattern(eventId);
  }

  private async checkCreatorQualityPattern(eventId: string): Promise<void> {
    const creatorQualityQuery = `
      SELECT
        AVG(cw.final_quality_score) as avg_quality,
        COUNT(*) as event_count
      FROM events e
      JOIN curation_workflows cw ON e.id = cw.event_id
      WHERE e.creator_id = (SELECT creator_id FROM events WHERE id = $1)
        AND e.end_time >= NOW() - INTERVAL '3 months'
        AND cw.final_quality_score IS NOT NULL
    `;

    const result = await this.db.query(creatorQualityQuery, [eventId]);

    if (result.length > 0 && result[0].event_count >= 3) {
      const avgQuality = parseFloat(result[0].avg_quality);

      if (avgQuality < 3.0) {
        await this.createCreatorQualityAlert(eventId, {
          averageQuality: avgQuality,
          eventCount: result[0].event_count
        });
      }
```

```
    }
  }

  private async createQualityAlert(eventId: string, alertType: string, metadata: any): Promise<void> {
    await this.db.query(`
      INSERT INTO quality_alerts (
        event_id, alert_type, metadata, status, created_at
      ) VALUES ($1, $2, $3, 'pending', NOW())
    `, [eventId, alertType, JSON.stringify(metadata)]);

    // Notify curation team
    await this.notificationService.sendToTeam({
      type: 'quality_alert',
      title: `Quality Alert: ${alertType}`,
      message: `Event ${eventId} has triggered a quality alert`,
      metadata: { eventId, alertType, ...metadata }
    });
  }
}
```

Reputation-Weighted Community Moderation

typescript

```
// Advanced Reputation System for Community Moderation
class ReputationWeightedModeration {
  constructor(private db: DatabaseService) {}

  async calculateCommunityConsensus(eventId: string): Promise<CommunityConsensus> {
    const consensusQuery = `
      WITH user_weights AS (
        SELECT
          cr.user_id,
          cr.rating,
          cr.accurate_description_score,
          cr.organization_score,
          cr.feedback,
          ur.community_score,
          ur.reviews_written,
          ur.reviews_helpful,
          -- Calculate user credibility weight
          CASE
            WHEN ur.community_score >= 800 THEN 2.0
            WHEN ur.community_score >= 600 THEN 1.5
            WHEN ur.community_score >= 400 THEN 1.0
            WHEN ur.community_score >= 200 THEN 0.7
            ELSE 0.5
          END *
          CASE
            WHEN ur.reviews_written = 0 THEN 0.5
            ELSE LEAST(2.0, (ur.reviews_helpful::float / ur.reviews_written) * 2.0)
          END as credibility_weight
        FROM community_reviews cr
        JOIN user_reputation ur ON cr.user_id = ur.user_id
        WHERE cr.event_id = $1
      ),
      weighted_consensus AS (
        SELECT
          SUM(rating * credibility_weight) / SUM(credibility_weight) as weighted_rating,
          SUM(accurate_description_score * credibility_weight) / SUM(credibility_weight) as weighted_accuracy,
          SUM(organization_score * credibility_weight) / SUM(credibility_weight) as weighted_organization,
          COUNT(*) as total_reviews,
          SUM(credibility_weight) as total_weight,
          STDDEV(rating) as rating_variance
        FROM user_weights
      )
      SELECT
```

```
      weighted_rating,
      weighted_accuracy,
      weighted_organization,
      total_reviews,
      total_weight,
      rating_variance,
      CASE
        WHEN rating_variance <= 1.0 AND total_reviews >= 5 THEN 'high'
        WHEN rating_variance <= 1.5 AND total_reviews >= 3 THEN 'medium'
        WHEN total_reviews >= 2 THEN 'low'
        ELSE 'insufficient'
      END as consensus_confidence
    FROM weighted_consensus
  `;

  const result = await this.db.query(consensusQuery, [eventId]);

  if (result.length === 0) {
    return {
      rating: null,
      accuracy: null,
      organization: null,
      confidence: 'insufficient',
      reviewCount: 0
    };
  }

  const row = result[0];
  return {
    rating: parseFloat(row.weighted_rating),
    accuracy: parseFloat(row.weighted_accuracy),
    organization: parseFloat(row.weighted_organization),
    confidence: row.consensus_confidence,
    reviewCount: row.total_reviews,
    variance: parseFloat(row.rating_variance)
  };
}

async detectAnomalousReviews(eventId: string): Promise<AnomalousReview[]> {
  // Use statistical analysis to detect outlier reviews
  const outlierQuery = `
    WITH review_stats AS (
      SELECT
        AVG(rating) as avg_rating,
```

```
      STDDEV(rating) as stddev_rating,
      COUNT(*) as review_count
    FROM community_reviews
    WHERE event_id = $1
  ),
  flagged_reviews AS (
    SELECT
      cr.id,
      cr.user_id,
      cr.rating,
      cr.feedback,
      ur.community_score,
      ur.reviews_written,
      ABS(cr.rating - rs.avg_rating) as deviation_from_mean,
      CASE
        WHEN ABS(cr.rating - rs.avg_rating) > (2 * rs.stddev_rating) THEN 'statistical_outlier'
        WHEN LENGTH(cr.feedback) < 20 AND ur.reviews_written < 5 THEN 'low_effort'
        WHEN ur.community_score < 100 AND ABS(cr.rating - rs.avg_rating) > rs.stddev_rating THEN 'low_credibi
        ELSE null
      END as anomaly_type
    FROM community_reviews cr
    JOIN user_reputation ur ON cr.user_id = ur.user_id
    CROSS JOIN review_stats rs
    WHERE cr.event_id = $1
      AND rs.review_count >= 5  -- Need sufficient reviews for statistical analysis
  )
  SELECT *
  FROM flagged_reviews
  WHERE anomaly_type IS NOT NULL
  ORDER BY deviation_from_mean DESC
`;

const anomalies = await this.db.query(outlierQuery, [eventId]);

return anomalies.map(row => ({
  reviewId: row.id,
  userId: row.user_id,
  anomalyType: row.anomaly_type,
  deviationScore: parseFloat(row.deviation_from_mean),
  userCredibility: row.community_score,
  confidence: this.calculateAnomalyConfidence(row)
}));
}
```

```
  private calculateAnomalyConfidence(anomaly: any): number {
    let confidence = 0.5;

    // Higher confidence for statistical outliers
    if (anomaly.anomaly_type === 'statistical_outlier') {
      confidence += 0.3;
    }

    // Factor in user's credibility
    if (anomaly.community_score < 100) {
      confidence += 0.2;
    }

    // Factor in deviation magnitude
    confidence += Math.min(0.3, anomaly.deviation_from_mean * 0.1);

    return Math.min(1.0, confidence);
  }
}
```

---

4. Workflow Orchestration & State Management

State Machine Implementation

typescript

```typescript
// Curation Workflow State Machine
enum CurationState {
  SUBMITTED = 'submitted',
  AI_SCREENING = 'ai_screening',
  HUMAN_REVIEW_PENDING = 'human_review_pending',
  HUMAN_REVIEW_IN_PROGRESS = 'human_review_in_progress',
  REVISION_REQUIRED = 'revision_required',
  APPROVED = 'approved',
  REJECTED = 'rejected',
  PUBLISHED = 'published',
  COMMUNITY_OVERSIGHT = 'community_oversight',
  FLAGGED = 'flagged',
  SUSPENDED = 'suspended'
}

enum CurationEvent {
  SUBMIT_EVENT = 'submit_event',
  AI_SCREENING_COMPLETE = 'ai_screening_complete',
  CURATOR_ASSIGNED = 'curator_assigned',
  REVIEW_SUBMITTED = 'review_submitted',
  REVISION_SUBMITTED = 'revision_submitted',
  COMMUNITY_FLAG = 'community_flag',
  QUALITY_ALERT = 'quality_alert',
  MANUAL_OVERRIDE = 'manual_override'
}

interface StateTransition {
  from: CurationState;
  to: CurationState;
  event: CurationEvent;
  conditions?: (context: WorkflowContext) => boolean;
  actions?: (context: WorkflowContext) => Promise<void>;
}

class CurationStateMachine {
  private transitions: StateTransition[] = [
    {
      from: CurationState.SUBMITTED,
      to: CurationState.AI_SCREENING,
      event: CurationEvent.SUBMIT_EVENT,
      actions: this.triggerAIScreening
    },
    {
```

```
    from: CurationState.AI_SCREENING,
    to: CurationState.HUMAN_REVIEW_PENDING,
    event: CurationEvent.AI_SCREENING_COMPLETE,
    conditions: (ctx) => ctx.aiResult.overallScore >= 0.6,
    actions: this.queueForHumanReview
  },
  {
    from: CurationState.AI_SCREENING,
    to: CurationState.REJECTED,
    event: CurationEvent.AI_SCREENING_COMPLETE,
    conditions: (ctx) => ctx.aiResult.overallScore < 0.3,
    actions: this.autoReject
  },
  {
    from: CurationState.HUMAN_REVIEW_PENDING,
    to: CurationState.HUMAN_REVIEW_IN_PROGRESS,
    event: CurationEvent.CURATOR_ASSIGNED,
    actions: this.assignCurator
  },
  {
    from: CurationState.HUMAN_REVIEW_IN_PROGRESS,
    to: CurationState.APPROVED,
    event: CurationEvent.REVIEW_SUBMITTED,
    conditions: (ctx) => ctx.reviewDecision === 'approve',
    actions: this.approveEvent
  },
  {
    from: CurationState.HUMAN_REVIEW_IN_PROGRESS,
    to: CurationState.REVISION_REQUIRED,
    event: CurationEvent.REVIEW_SUBMITTED,
    conditions: (ctx) => ctx.reviewDecision === 'needs_revision',
    actions: this.requestRevision
  },
  {
    from: CurationState.APPROVED,
    to: CurationState.PUBLISHED,
    event: CurationEvent.REVIEW_SUBMITTED,
    actions: this.publishEvent
  },
  {
    from: CurationState.PUBLISHED,
    to: CurationState.COMMUNITY_OVERSIGHT,
    event: CurationEvent.REVIEW_SUBMITTED,
    actions: this.enableCommunityOversight
```

```
  }
];

async processEvent(
  workflowId: string,
  event: CurationEvent,
  context: WorkflowContext
): Promise<CurationState> {
  const currentState = await this.getCurrentState(workflowId);

  const validTransitions = this.transitions.filter(
    t => t.from === currentState && t.event === event
  );

  if (validTransitions.length === 0) {
    throw new Error(`Invalid transition: ${currentState} -> ${event}`);
  }

  // Find the first transition that meets conditions
  const transition = validTransitions.find(t =>
    !t.conditions || t.conditions(context)
  );

  if (!transition) {
    throw new Error(`No valid transition found for current context`);
  }

  // Execute transition actions
  if (transition.actions) {
    await transition.actions(context);
  }

  // Update state
  await this.updateState(workflowId, transition.to, context);

  // Log state change
  await this.logStateChange(workflowId, currentState, transition.to, event, context);

  return transition.to;
}

private async triggerAIScreening(context: WorkflowContext): Promise<void> {
  // Queue AI screening job
  await this.jobQueue.add('ai-screening', {
```

```
      eventId: context.eventId,
      workflowId: context.workflowId
    });
  }

  private async queueForHumanReview(context: WorkflowContext): Promise<void> {
    // Calculate priority and add to curator queue
    const priority = this.calculateReviewPriority(context);

    await this.db.query(`
      UPDATE curation_workflows
      SET current_stage = 'human_review',
          ai_screening_completed_at = NOW(),
          ai_screening_passed = true,
          priority_score = $2
      WHERE id = $1
    `, [context.workflowId, priority]);

    // Notify available curators
    await this.notifyCurators(context.workflowId, priority);
  }

  private async autoReject(context: WorkflowContext): Promise<void> {
    await this.db.query(`
      UPDATE curation_workflows
      SET overall_status = 'rejected',
          rejection_reason = 'Failed AI quality screening',
          completed_at = NOW()
      WHERE id = $1
    `, [context.workflowId]);

    // Send feedback to creator
    await this.sendCreatorFeedback(context, 'rejected');
  }
}
```

Job Queue & Background Processing

typescript

```typescript
// Background Job Processing for Curation Pipeline
import Bull from 'bull';
import { RedisClient } from './redis-client';

interface CurationJob {
  eventId: string;
  workflowId: string;
  priority?: number;
  attempts?: number;
  metadata?: any;
}

class CurationJobProcessor {
  private aiScreeningQueue: Bull.Queue;
  private notificationQueue: Bull.Queue;
  private analyticsQueue: Bull.Queue;

  constructor(private redis: RedisClient) {
    this.aiScreeningQueue = new Bull('ai-screening', {
      redis: redis.connectionOptions,
      defaultJobOptions: {
        removeOnComplete: 100,
        removeOnFail: 50,
        attempts: 3,
        backoff: {
          type: 'exponential',
          delay: 2000
        }
      }
    });

    this.setupJobProcessors();
  }

  private setupJobProcessors(): void {
    // AI Screening Job Processor
    this.aiScreeningQueue.process('ai-screening', 5, async (job) => {
      const { eventId, workflowId } = job.data as CurationJob;

      try {
        // Update job progress
        await job.progress(10);
```

```
    // Fetch event data
    const eventData = await this.getEventData(eventId);
    await job.progress(20);

    // Run AI screening
    const aiResult = await this.aiScreeningService.screenEvent(eventData);
    await job.progress(80);

    // Store results
    await this.storeAIResults(workflowId, aiResult);
    await job.progress(90);

    // Trigger next workflow step
    await this.stateMachine.processEvent(
      workflowId,
      CurationEvent.AI_SCREENING_COMPLETE,
      { workflowId, eventId, aiResult }
    );

    await job.progress(100);

    return { success: true, aiResult };

  } catch (error) {
    // Log error and update workflow
    await this.handleJobError(workflowId, 'ai_screening_failed', error);
    throw error;
  }
});

// Notification Job Processor
this.notificationQueue.process('send-notification', 10, async (job) => {
  const { type, recipients, content } = job.data;

  await this.notificationService.sendBulk({
    type,
    recipients,
    content,
    priority: job.opts.priority
  });
});

// Analytics Job Processor
this.analyticsQueue.process('track-curation-event', 20, async (job) => {
```

```
    const { eventType, data } = job.data;

    await this.analyticsService.track({
      type: `curation_${eventType}`,
      data,
      timestamp: new Date()
    });
  });

  // Job monitoring and alerting
  this.setupJobMonitoring();
}

private setupJobMonitoring(): void {
  this.aiScreeningQueue.on('failed', async (job, err) => {
    console.error(`AI Screening job ${job.id} failed:`, err);

    // Alert on repeated failures
    const failureCount = await this.getJobFailureCount(job.data.workflowId);
    if (failureCount >= 3) {
      await this.alertCurationTeam({
        type: 'repeated_job_failures',
        workflowId: job.data.workflowId,
        error: err.message
      });
    }
  });

  this.aiScreeningQueue.on('stalled', async (job) => {
    console.warn(`AI Screening job ${job.id} stalled`);

    // Auto-retry stalled jobs
    await job.retry();
  });

  // Queue health monitoring
  setInterval(async () => {
    const health = await this.getQueueHealth();
    if (health.aiScreeningBacklog > 100) {
      await this.alertCurationTeam({
        type: 'queue_backlog_high',
        queue: 'ai-screening',
        backlog: health.aiScreeningBacklog
      });
```

```
    }
  }, 60000); // Check every minute
}

async addAIScreeningJob(eventId: string, workflowId: string, priority = 0): Promise<void> {
  await this.aiScreeningQueue.add('ai-screening', {
    eventId,
    workflowId,
    priority
  }, {
    priority: priority,
    delay: priority > 50 ? 0 : 5000 // High priority jobs run immediately
  });
}

async getQueueHealth(): Promise<QueueHealth> {
  const [aiWaiting, aiActive, aiFailed] = await Promise.all([
    this.aiScreeningQueue.getWaiting(),
    this.aiScreeningQueue.getActive(),
    this.aiScreeningQueue.getFailed()
  ]);

  return {
    aiScreeningBacklog: aiWaiting.length,
    aiScreeningActive: aiActive.length,
    aiScreeningFailed: aiFailed.length,
    overallHealth: this.calculateOverallHealth({
      waiting: aiWaiting.length,
      active: aiActive.length,
      failed: aiFailed.length
    })
  };
}
}
```

---

5. Performance Optimization & Monitoring

Curation Pipeline Metrics

typescript

```
// Comprehensive Curation Metrics & Analytics
interface CurationMetrics {
  throughput: {
    eventsSubmittedPerHour: number;
    eventsProcessedPerHour: number;
    averageProcessingTime: number;
    bottleneckStage: string;
```