```python
import streamlit as st
import requests
import time
import os
import re
import subprocess
import json
from datetime import datetime
from typing import Dict, List, Tuple, Any
import pandas as pd
import tempfile
from moviepy.editor import VideoFileClip, concatenate_videoclips
import shutil
import zipfile
import io

# =========================
# CONFIGURATION
# =========================
PERPLEXITY_API_KEY = "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"
PERPLEXITY_MODEL = "sonar-pro"
GEMINI_API_KEY = "AIzaSyCaZBQcouetlGMzIl6fsfbGcgeOpxuGHAY"

# Enhanced directory structure
VIDEO_OUTPUT_FOLDER = "video_chunks"
TRIMMED_VIDEO_FOLDER = "trimmed_video_chunks"
ANALYSIS_OUTPUT_FOLDER = "feature_analysis"
DOCUMENTATION_FOLDER = "veo3_documentation"
QA_REPORTS_FOLDER = "quality_reports"

for folder in [VIDEO_OUTPUT_FOLDER, TRIMMED_VIDEO_FOLDER, ANALYSIS_OUTPUT_FOLDER,
               DOCUMENTATION_FOLDER, QA_REPORTS_FOLDER]:
    os.makedirs(folder, exist_ok=True)

# =========================
# ENHANCED CAMERA INSTRUCTION WITH PRECISE TIMING
# =========================
EXACT_CAMERA_TEXT = ("Start behind a partial wall or plant on the left side. "
                    "Slowly dolly from left to right across the scene, passing behind objects mid-shot. "
                    "The video MUST END exactly as the camera becomes fully concealed behind the right-side object. "
                    "Duration: 6.5 seconds total - spend 5.5 seconds moving across scene, then 1 second becoming concealed. "
                    "NO FOOTAGE after camera is hidden behind right object. "
                    "Maintain smooth, steady motion throughout.")

# =========================
# ENHANCED FEATURE ANALYSIS ENGINE
# =========================
class EnhancedFeatureAnalysisEngine:
    """Advanced feature analysis engine that processes detailed feature descriptions."""

    def __init__(self):
        self.analysis_cache = {}

    def parse_feature_description(self, feature_description: str) -> Dict[str, Any]:
        """Parse and extract structured insights from detailed feature description."""

        # Enhanced analysis prompt for detailed feature processing
        analysis_prompt = f"""
        You are an expert product analyst specializing in feature breakdown and stakeholder mapping.

        Analyze this DETAILED FEATURE DESCRIPTION and extract comprehensive insights:

        FEATURE DESCRIPTION:
        {feature_description}

        EXTRACT THE FOLLOWING INFORMATION (JSON format):
        {{
            "feature_name": "Extract the feature name",
            "feature_category": "Classify the feature type (e.g., 'User Engagement Tool', 'Analytics Platform', etc.)",
            "core_value_propositions": [
                "Primary value proposition 1",
                "Primary value proposition 2",
                "Primary value proposition 3"
            ],
            "customer_pain_points": [
                "Specific pain point this feature solves 1",
                "Specific pain point this feature solves 2",
                "Specific pain point this feature solves 3",
                "Specific pain point this feature solves 4"
            ],
            "key_capabilities": [
                "Technical capability 1",
                "Technical capability 2",
                "Technical capability 3"
            ],
            "quantified_benefits": [
                "Measurable benefit 1 (include numbers if provided)",
                "Measurable benefit 2 (include numbers if provided)",
                "Measurable benefit 3 (include numbers if provided)"
            ],
            "target_user_personas": [
                {{
                    "persona_name": "Primary User Type",
                    "role": "Their job role/position",
                    "current_challenges": "What they struggle with currently",
                    "desired_outcomes": "What success looks like for them",
                    "usage_context": "When/where they use this feature"
                }},
                {{
                    "persona_name": "Secondary User Type",
                    "role": "Their job role/position",
                    "current_challenges": "What they struggle with currently",
                    "desired_outcomes": "What success looks like for them",
                    "usage_context": "When/where they use this feature"
                }}
            ],
            "stakeholder_journeys": {{
                "primary_user": {{
                    "persona": "Primary user description with role",
                    "current_state": "How they currently handle this problem/need",
                    "pain_points": ["Current frustration 1", "Current frustration 2"],
                    "discovery_trigger": "What makes them look for a solution",
                    "evaluation_criteria": ["What they care about when choosing", "Success metric 1", "Success metric 2"],
                    "adoption_barriers": ["Potential concern 1", "Potential concern 2"],
                    "desired_state": "Their ideal outcome with this feature",
                    "success_indicators": ["How they measure success 1", "How they measure success 2"],
                    "emotional_arc": "Frustration в†' Hope в†' Confidence в†' Success"
                }},
                "secondary_stakeholders": [
                    {{
                        "role": "Secondary stakeholder role",
                        "relationship_to_primary": "How they relate to primary user",
                        "impact": "How this feature affects them",
                        "concerns": "Their potential concerns or needs",
                        "success_criteria": "What success means for them"
                    }}

            }},
            "use_cases": [
                {{
                    "scenario": "Use case scenario 1",
                    "context": "When/where this happens",
                    "user_goal": "What user wants to achieve",
                    "feature_role": "How feature helps achieve goal",
                    "outcome": "Expected result"
                }},
                {{
                    "scenario": "Use case scenario 2",
                    "context": "When/where this happens",
                    "user_goal": "What user wants to achieve",
                    "feature_role": "How feature helps achieve goal",
                    "outcome": "Expected result"
                }},
                {{
                    "scenario": "Use case scenario 3",
                    "context": "When/where this happens",
                    "user_goal": "What user wants to achieve",
                    "feature_role": "How feature helps achieve goal",
                    "outcome": "Expected result"
                }}
            ],
            "business_impact": {{
                "efficiency_gains": "How this improves efficiency (with numbers if available)",
                "cost_savings": "Potential cost reductions (with numbers if available)",
                "revenue_opportunities": "Revenue impact (with numbers if available)",
                "competitive_advantages": "Market differentiation points",
                "roi_indicators": ["ROI metric 1", "ROI metric 2", "ROI metric 3"]
            }},
            "technical_requirements": [
                "Key technical requirement 1",
                "Key technical requirement 2",
                "Integration requirement 1",
                "Performance requirement 1"
            ],
            "success_metrics": [
                "Quantifiable outcome metric 1",
                "User behavior metric 1",
                "Business impact metric 1",
                "Adoption metric 1"
            ],
            "narrative_themes": {{
                "central_conflict": "Main problem/challenge this feature addresses",
                "resolution_path": "How feature resolves the conflict",
                "transformation_story": "The before/after user transformation",
                "emotional_beats": [
                    "Current frustration/struggle",
                    "Problem escalation/pain",
                    "Discovery of solution",
                    "Initial hope/curiosity",
                    "Engagement/trial",
                    "Growing confidence",
                    "Transformation/success",
                    "Mastery/celebration"
                ],
                "visual_metaphors": [
                    "Metaphor representing current state",
                    "Metaphor representing transformation",
                    "Metaphor representing success state"
                ],
                "story_settings": [
                    "Real-world environment 1 where feature is used",
                    "Real-world environment 2 where feature is used",
                    "Real-world environment 3 where feature is used"

            }},
            "competitive_differentiation": {{
                "unique_advantages": ["What makes this feature different 1", "What makes this feature different 2"],
                "market_gaps": ["Gap this fills 1", "Gap this fills 2"],
                "user_preference_drivers": ["Why users choose this 1", "Why users choose this 2"]
            }}
        }}

        IMPORTANT: Provide ONLY the JSON response. Extract specific details from the feature description provided. If quantified benefits or metrics are mentioned, inc
        """

        url = "https://api.perplexity.ai/chat/completions"
        headers = {
            "Authorization": f"Bearer {PERPLEXITY_API_KEY}",
            "Content-Type": "application/json"
        }

        payload = {
            "model": PERPLEXITY_MODEL,
            "messages": [
                {"role": "system", "content": "You are an expert feature analyst who extracts structured insights from feature descriptions."},
                {"role": "user", "content": analysis_prompt}
            ],
            "max_tokens": 3000,
            "temperature": 0.5
        }

        try:
            response = requests.post(url, headers=headers, json=payload)
            response.raise_for_status()
            analysis_text = response.json()['choices'][0]['message']['content']

            # Extract JSON from response
            json_match = re.search(r'\{.*\}', analysis_text, re.DOTALL)
            if json_match:
                analysis_data = json.loads(json_match.group())

                # Save analysis with timestamp
                timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
                analysis_file = os.path.join(ANALYSIS_OUTPUT_FOLDER, f"feature_analysis_{timestamp}.json")
                with open(analysis_file, 'w', encoding='utf-8') as f:
                    json.dump(analysis_data, f, indent=2)

                # Add metadata
                analysis_data['_metadata'] = {
                    'analysis_timestamp': timestamp,
                    'original_description': feature_description,
                    'analysis_file': analysis_file
                }

                return analysis_data
            else:
                st.error("Could not extract JSON from analysis response")
                st.text_area("Raw Analysis Response:", analysis_text, height=300)
                return self._get_fallback_analysis(feature_description)

        except json.JSONDecodeError as e:
            st.error(f"JSON parsing error: {str(e)}")
            st.text_area("Invalid JSON Response:", analysis_text, height=300)
            return self._get_fallback_analysis(feature_description)
        except Exception as e:
            st.error(f"Feature analysis failed: {str(e)}")
            return self._get_fallback_analysis(feature_description)

    def _get_fallback_analysis(self, feature_description: str) -> Dict[str, Any]:
        """Generate fallback analysis when API fails."""
        return {
            "feature_name": "Feature Analysis Failed - Using Fallback",
            "feature_category": "General Feature",
            "core_value_propositions": [
                "Improves user experience and workflow efficiency",
                "Provides better tools and capabilities",
                "Delivers measurable business value"
            ],
            "customer_pain_points": [
                "Current process is manual and time-consuming",
                "Users lack proper tools for the task",
                "Existing solutions are inadequate",
                "Manual work leads to errors and inefficiencies"
            ],
            "key_capabilities": [
                "Core functionality 1",
                "Core functionality 2",
                "Integration capabilities"
            ],
            "quantified_benefits": [
                "Improves efficiency significantly",
                "Reduces manual work substantially",
                "Increases user satisfaction"
            ],
            "target_user_personas": [
                {
                    "persona_name": "Primary User",
                    "role": "Professional seeking efficiency",
                    "current_challenges": "Manual, inefficient processes",
                    "desired_outcomes": "Automated, streamlined workflows",
                    "usage_context": "Daily work environment"
                }
            ],
            "stakeholder_journeys": {
                "primary_user": {
                    "persona": "Professional user seeking efficiency",
                    "current_state": "Manual, time-consuming processes",
                    "pain_points": ["Too much manual work", "Prone to errors"],
                    "discovery_trigger": "Need for better efficiency",
                    "evaluation_criteria": ["Ease of use", "Time savings", "Reliability"],
                    "adoption_barriers": ["Learning curve", "Change resistance"],
                    "desired_state": "Automated, efficient workflows",
                    "success_indicators": ["Time saved", "Error reduction", "User satisfaction"],
                    "emotional_arc": "Frustration в†’ Hope в†’ Confidence в†’ Success"
                },
                "secondary_stakeholders": []
            },
            "use_cases": [
                {
                    "scenario": "Daily workflow improvement",
                    "context": "Regular work environment",
                    "user_goal": "Complete tasks efficiently",
                    "feature_role": "Automates and streamlines process",
                    "outcome": "Faster, more accurate results"
                }
            ],
            "business_impact": {
                "efficiency_gains": "Significant workflow improvements",
                "cost_savings": "Reduced manual labor costs",
                "revenue_opportunities": "Improved productivity leads to revenue gains",
                "competitive_advantages": "Better tools than competitors",
                "roi_indicators": ["Time saved", "Error reduction", "User adoption"]
            },
            "narrative_themes": {
                "central_conflict": "Inefficient current process",
                "resolution_path": "Feature provides solution",
                "transformation_story": "From manual struggle to automated success",
                "emotional_beats": [
                    "Current frustration", "Problem escalation", "Discovery", "Hope",
                    "Engagement", "Growing confidence", "Transformation", "Success"
                ],
                "visual_metaphors": ["Journey from chaos to order", "Transformation", "Achievement"],
                "story_settings": ["Office environment", "Professional workspace", "Meeting room"]
            },
            "_metadata": {
                "analysis_timestamp": datetime.now().strftime("%Y%m%d_%H%M%S"),
                "original_description": feature_description,
                "is_fallback": True
            }
        }

    def validate_feature_description(self, feature_description: str) -> Dict[str, Any]:
        """Validate that feature description contains sufficient detail."""
        validation_result = {
            "is_valid": False,
            "completeness_score": 0,
            "missing_elements": [],
            "recommendations": []
        }

        # Check for key elements
        required_elements = {
            "capabilities": ["capabilit", "feature", "function", "enable", "provide"],
            "benefits": ["benefit", "improve", "increase", "reduce", "save"],
            "use_cases": ["use case", "scenario", "example", "application"],
            "metrics": ["%", "percent", "times", "x", "increase", "decrease", "improve"]
        }

        description_lower = feature_description.lower()
        found_elements = {}

        for element, keywords in required_elements.items():
            found_elements[element] = any(keyword in description_lower for keyword in keywords)
            if found_elements[element]:
                validation_result["completeness_score"] += 25
            else:
                validation_result["missing_elements"].append(element)

        # Check length
```

```python
        if len(feature_description) < 200:
            validation_result["recommendations"].append("Feature description should be more detailed (aim for 200+ characters)")

        # Check for specific sections
        if not any(section in description_lower for section in ["capabilities:", "benefits:", "use cases:"]):
            validation_result["recommendations"].append("Consider organizing with clear sections: Capabilities, Benefits, Use Cases")

        # Determine validity
        validation_result["is_valid"] = validation_result["completeness_score"] >= 50

        return validation_result

# =========================
# NARRATIVE EXCELLENCE ENGINE (Enhanced)
# =========================
class NarrativeExcellenceEngine:
    """Advanced narrative and emotional arc management with feature integration."""

    def __init__(self, feature_analysis: Dict[str, Any]):
        self.feature_analysis = feature_analysis
        self.emotional_beats = feature_analysis.get('narrative_themes', {}).get('emotional_beats',
                                                     ["Struggle", "Escalation", "Discovery", "Hope", "Engagement", "Growth", "Transformation", "Success"])

    def generate_emotional_arc_structure(self) -> Dict[str, Any]:
        """Generate 8-chunk emotional progression structure based on feature analysis."""

        # Map feature-driven emotional beats to 8 chunks
        emotional_progression = {}
        beats = self.emotional_beats

        for i in range(8):
            chunk_num = i + 1
            beat_index = min(i, len(beats) - 1)

            emotional_progression[chunk_num] = {
                "beat": beats[beat_index] if i < len(beats) else "Success continuation",
                "emotion": self._map_beat_to_emotion(beats[beat_index] if i < len(beats) else "Success"),
                "intensity": self._calculate_intensity(chunk_num),
                "feature_focus": self._get_feature_focus(chunk_num),
                "stakeholder_stage": self._get_stakeholder_stage(chunk_num)
            }

        return {
            "emotional_progression": emotional_progression,
            "narrative_themes": self.feature_analysis.get('narrative_themes', {}),
            "stakeholder_journey": self.feature_analysis.get('stakeholder_journeys', {}),
            "use_cases": self.feature_analysis.get('use_cases', []),
            "visual_metaphors": self.feature_analysis.get('narrative_themes', {}).get('visual_metaphors', [])
        }

    def _map_beat_to_emotion(self, beat: str) -> str:
        """Map narrative beat to specific emotion."""
        beat_emotions = {
            "current frustration": "Frustration/Stress",
            "problem escalation": "Increased Frustration",
            "discovery of solution": "Hope/Curiosity",
            "initial hope": "Cautious Optimism",
            "engagement": "Growing Interest",
            "growing confidence": "Confidence",
            "transformation": "Satisfaction/Relief",
            "success": "Joy/Achievement"
        }

        beat_lower = beat.lower()
        for key, emotion in beat_emotions.items():
            if key in beat_lower:
                return emotion

        return "Neutral"

    def _calculate_intensity(self, chunk_num: int) -> int:
        """Calculate emotional intensity for chunk (1-10 scale)."""
        intensity_curve = [3, 4, 5, 6, 7, 8, 9, 10]  # Builds to climax
        return intensity_curve[min(chunk_num - 1, 7)]

    def _get_feature_focus(self, chunk_num: int) -> str:
        """Get feature aspect to focus on for each chunk."""
        feature_focuses = [
            "Problem identification",
            "Pain point escalation",
            "Feature discovery",
            "Initial capability exploration",
            "Active feature use",
            "Advanced capabilities",
            "Full transformation",
            "Success demonstration"
        ]
        return feature_focuses[min(chunk_num - 1, 7)]

    def _get_stakeholder_stage(self, chunk_num: int) -> str:
        """Get stakeholder journey stage for chunk."""
        primary_journey = self.feature_analysis.get('stakeholder_journeys', {}).get('primary_user', {})
        journey_stages = primary_journey.get('journey_stages',
                                            ["Problem awareness", "Solution search", "Evaluation", "Trial",
                                             "Adoption", "Optimization", "Mastery", "Advocacy"])

        return journey_stages[min(chunk_num - 1, len(journey_stages) - 1)]

# =========================
# ENHANCED STORY GENERATION WITH FEATURE INTEGRATION
# =========================
def generate_feature_driven_story_chunks(feature_analysis: Dict[str, Any],
                                       narrative_structure: Dict[str, Any]) -> str:
    """Generate 8 chunks with comprehensive feature analysis integration and dramatic emotional journey."""

    url = "https://api.perplexity.ai/chat/completions"
    headers = {
        "Authorization": f"Bearer {PERPLEXITY_API_KEY}",
        "Content-Type": "application/json"
    }

    # Extract comprehensive feature data
    feature_name = feature_analysis.get('feature_name', 'Feature')
    core_values = feature_analysis.get('core_value_propositions', [])
    pain_points = feature_analysis.get('customer_pain_points', [])
    capabilities = feature_analysis.get('key_capabilities', [])
    benefits = feature_analysis.get('quantified_benefits', [])
    use_cases = feature_analysis.get('use_cases', [])
    stakeholder_journey = feature_analysis.get('stakeholder_journeys', {}).get('primary_user', {})
    narrative_themes = feature_analysis.get('narrative_themes', {})
    emotional_progression = narrative_structure.get('emotional_progression', {})

    # Create comprehensive story generation prompt with dramatic emotional journey
    enhanced_story_prompt = f"""
You are creating a DRAMATIC FEATURE TRANSFORMATION STORY that follows a complete emotional journey from devastation to triumph, while showcasing the complete value and

==== MANDATORY 8-CHUNK DRAMATIC STORY ARC ====

CHUNK 1: **MANUAL STRUGGLE ACTIVITY** - Character doing frustrating manual work (filing papers, manual data entry, phone calls, physical sorting)

CHUNK 2: **CRISIS ESCALATION ACTIVITY** - Character dealing with urgent problems (emergency meeting, rushing with documents, fixing broken process)

CHUNK 3: **DISCOVERY ACTIVITY** - Character researching solutions (reading, browsing, asking colleagues, investigating options)

CHUNK 4: **TRIAL ACTIVITY** - Character cautiously testing new approach (setup, configuration, first attempt, careful observation)

CHUNK 5: **BREAKTHROUGH ACTIVITY** - Character experiencing success (celebrating, sharing results, demonstrating to others)

CHUNK 6: **MASTERY ACTIVITY** - Character becoming expert (training others, optimizing, scaling up usage)

CHUNK 7: **SUCCESS DEMONSTRATION** - Character presenting achievements (client meeting, presentation, showing results)

CHUNK 8: **FUTURE PLANNING** - Character planning expansion (strategy session, road mapping, mentoring others)

==== ACTIVITY VARIETY ENFORCEMENT ====

CRITICAL: Each chunk must show COMPLETELY DIFFERENT professional activities:
- NO repetitive dashboard viewing across multiple chunks
- NO similar screen-checking activities
- Each scene shows UNIQUE work scenarios and interactions
- Vary between: meetings, presentations, hands-on work, collaboration, planning, problem-solving
- Different locations: office, conference room, workshop, client site, cafe, home office

==== FEATURE ANALYSIS INTEGRATION ====

FEATURE NAME: {feature_name}
CATEGORY: {feature_analysis.get('feature_category', 'Business Tool')}

CORE VALUE PROPOSITIONS:
{chr(10).join(f"вЂў {value}" for value in core_values[:3])}

CUSTOMER PAIN POINTS TO DEMONSTRATE:
{chr(10).join(f"вЂў {pain}" for pain in pain_points[:4])}

KEY CAPABILITIES TO SHOWCASE:
{chr(10).join(f"вЂў {cap}" for cap in capabilities[:3])}

QUANTIFIED BENEFITS TO HIGHLIGHT:
{chr(10).join(f"вЂў {benefit}" for benefit in benefits[:3])}

==== STAKEHOLDER JOURNEY INTEGRATION ====

PRIMARY USER PERSONA: {stakeholder_journey.get('persona', 'Professional user')}
CURRENT STATE: {stakeholder_journey.get('current_state', 'Manual processes')}
PAIN POINTS: {', '.join(stakeholder_journey.get('pain_points', [])[:2])}
DESIRED STATE: {stakeholder_journey.get('desired_state', 'Efficient automated workflow')}
SUCCESS INDICATORS: {', '.join(stakeholder_journey.get('success_indicators', [])[:2])}
EMOTIONAL ARC: {stakeholder_journey.get('emotional_arc', 'Challenge в†’ Success')}

==== USE CASE INTEGRATION ====
{chr(10).join(f"USE CASE {i+1}: {case.get('scenario', 'N/A')} - {case.get('outcome', 'N/A')}" for i, case in enumerate(use_cases[:3]))}

==== NARRATIVE THEMES ====
CENTRAL CONFLICT: {narrative_themes.get('central_conflict', 'User challenge')}
RESOLUTION PATH: {narrative_themes.get('resolution_path', 'Feature solution')}
TRANSFORMATION STORY: {narrative_themes.get('transformation_story', 'Before/after improvement')}
VISUAL METAPHORS: {', '.join(narrative_themes.get('visual_metaphors', [])[:3])}
STORY SETTINGS: {', '.join(narrative_themes.get('story_settings', [])[:3])}

==== EMOTIONAL PROGRESSION MAP ====
{json.dumps(emotional_progression, indent=2)}

==== CINEMATIC STORY REQUIREMENTS ====

1. **FEATURE VALUE DEMONSTRATION**: Each chunk must progressively demonstrate specific feature capabilities and their impact on real user pain points.

2. **STAKEHOLDER JOURNEY ALIGNMENT**: Follow the primary user's complete transformation from current frustrating state to desired successful state.

3. **USE CASE INTEGRATION**: Incorporate real use cases from the feature analysis into scene contexts.

4. **EMOTIONAL ARC MASTERY**: Hit precise emotional beats that build authentic engagement and show genuine transformation.

5. **QUANTIFIED IMPACT SHOWCASE**: Where possible, visually represent the quantified benefits and improvements.

6. **PROGRESSIVE CAPABILITY REVEAL**: Each scene should reveal new aspects of the feature's capabilities and value.

7. **AUTHENTIC PROFESSIONAL CONTEXT**: Show the feature in realistic professional environments and workflows.

8. **DRAMATIC EMOTIONAL STORYTELLING**: Each chunk must show authentic human emotion and personal stakes, making viewers feel the character's journey from devastation

9. **ACTIVITY DIVERSITY MANDATE**: Each chunk must demonstrate the feature through COMPLETELY different professional activities and scenarios:
   - Chunk 1: Manual/traditional work methods
   - Chunk 2: Crisis management/urgent situations
   - Chunk 3: Research/discovery activities
   - Chunk 4: Initial testing/setup activities
   - Chunk 5: Active feature usage/breakthrough moments
   - Chunk 6: Advanced usage/optimization activities
   - Chunk 7: Results presentation/demonstration activities
   - Chunk 8: Strategic planning/mentoring activities

10. **VISUAL VARIETY REQUIREMENTS**: Ensure each scene shows different:
    - Physical activities (typing vs. presenting vs. collaborating vs. analyzing)
    - Work tools (computers vs. whiteboards vs. documents vs. mobile devices)
    - Professional interactions (solo work vs. team meetings vs. client presentations)
    - Environmental settings (office desk vs. conference room vs. workshop vs. client site)


==== ACTIVITY EXAMPLES FOR EACH CHUNK ====

CHUNK 1 Examples:
- Manually sorting through physical feedback forms
- Making individual phone calls to collect opinions
- Typing survey responses into spreadsheets by hand
- Searching through email chains for feedback

CHUNK 2 Examples:
- Rushing to prepare last-minute reports
- Emergency meeting due to lack of data
- Frantically calling clients for missing information
- Working late trying to compile manual reports

CHUNK 3 Examples:
- Reading articles about feedback solutions
- Having coffee conversation with colleague about tools
- Browsing software comparison websites
- Attending demo or webinar about new solutions
```

```
CHUNK 4 Examples:
- Setting up software for first time
- Creating first survey or feedback form
- Training team members on new process
- Testing with small group or pilot project

CHUNK 5 Examples:
- Receiving first real-time feedback notifications
- Watching live response analytics come in
- Sharing exciting results with team members
- Celebrating first successful campaign

CHUNK 6 Examples:
- Optimizing advanced features and settings
- Creating sophisticated automated workflows
- Training other departments on the system
- Scaling successful processes company-wide

CHUNK 7 Examples:
- Presenting impressive results to executives
- Demonstrating ROI to stakeholders
- Showing before/after comparisons to clients
- Receiving recognition or awards for improvements

CHUNK 8 Examples:
- Planning future feature implementations
- Mentoring new team members on best practices
- Discussing integration with other business tools
- Strategizing about expanding usage across organization


==== FORMAT FOR EACH CHUNK (8 TOTAL) ====

CHUNK X: [Emotional Story Title + Feature Capability - e.g., "The Breaking Point - Feedback Collection Crisis"]

CHARACTER DNA: [IDENTICAL character descriptions representing primary user persona - MUST BE EXACT SAME IN ALL CHUNKS]

ACTIVITY VARIATION REQUIREMENT: While the character remains identical, they must be shown in COMPLETELY different professional activities across the 8 chunks. Same per

[Include: Role, appearance, clothing, mannerisms, background - COPY EXACTLY TO ALL CHUNKS]
[MANDATE: Show this person doing VARIED professional activities - never repeat the same type of work task]

Primary User Persona: {stakeholder_journey.get('persona', 'Professional seeking efficiency')}
[Include: Role, appearance, clothing, mannerisms, background - COPY EXACTLY TO ALL CHUNKS]

[STAKEHOLDER CONTEXT]: [Current journey stage from emotional progression]

[PAIN POINT ADDRESSED]: [Specific pain point from analysis that this chunk addresses]

[FEATURE CAPABILITY]: [Specific capability being demonstrated in this scene]

[QUANTIFIED BENEFIT]: [Measurable benefit shown/implied in this scene]

[EMOTIONAL BEAT]: [Current emotional state with intensity level from progression map]

[USE CASE CONTEXT]: [Which use case scenario this relates to, if applicable]

[SETTING]: [UNIQUE professional environment from story settings - different from all other chunks - should reflect emotional state]

[CINEMATIC ACTION STORY]: [COMPLETE EMOTIONAL NARRATIVE SCENE - Write as compelling dramatic story with:
вЂў Character's internal emotional state and external situation
вЂў Rising dramatic tension or discovery moment
вЂў Feature interaction/implementation within the emotional context
вЂў Multiple connected actions showing both workflow and emotional journey
вЂў Visible transformation/results and character's emotional response
вЂў Scene conclusion with emotional impact and setup for next chunk
Write as engaging narrative prose - minimum 5-6 sentences telling the complete emotional and professional story of this moment. Show both the feature capabilities AND

[CAMERA]: {EXACT_CAMERA_TEXT}

[TIMING]: Duration: 6.5 seconds (5.5s movement + 1s concealment behind right object)

[VISUAL FEATURE DEMONSTRATION]: [How the feature's value is visually communicated without on-screen text]

[PROFESSIONAL CONTEXT]: [Realistic workplace/professional setting details]

[TRANSFORMATION ELEMENT]: [What specific improvement/change is visible in this scene]

[AUDIO]: [Professional ambient sounds, any natural dialogue that reinforces feature benefits and emotional state]

[VISUAL METAPHORS]: [Incorporate relevant metaphors from analysis that support emotional journey]

[STAKEHOLDER SUCCESS INDICATOR]: [How success is visually represented for this user persona]

[NARRATIVE PROGRESSION]: [How this chunk advances the complete feature demonstration story and emotional arc]

[EXCLUSIONS]: No on-screen text, captions, subtitles, characters never look at camera, no footage after concealment

==== SPECIFIC DRAMATIC STORY PROGRESSION ====

CHUNK 1: Character faces complete professional failure with current methods, devastated about career prospects, shows failed feedback collection attempts
CHUNK 2: Crisis escalates, everything falls apart, character questions abilities, manual processes completely break down
CHUNK 3: In desperation, character searches for solutions and discovers {feature_name} - first glimmer of hope
CHUNK 4: Character cautiously tries the feature, experiences small wins with real-time feedback, dares to hope
CHUNK 5: Major breakthrough! Feature works brilliantly, character amazed by instant engagement and data flow
CHUNK 6: Character gains mastery and confidence, sees professional life transforming through advanced capabilities
CHUNK 7: Complete success achieved, character celebrates transformation with comprehensive analytics and recognition
CHUNK 8: Character becomes mentor/advocate, discusses future enhancements and inspires others

==== CRITICAL SUCCESS REQUIREMENTS ====

- Show REAL professional scenarios where this feature delivers value
- Demonstrate ACTUAL capabilities from the feature analysis
- Follow AUTHENTIC stakeholder journey progression
- Include SPECIFIC use case contexts from analysis
- Show MEASURABLE improvements and transformation
- Maintain IDENTICAL character DNA (same person throughout story)
- Each scene shows COMPLETELY DIFFERENT professional environment
- Feature benefits must be VISUALLY CLEAR without text overlays
- Emotional progression must feel GENUINE and earned with real human stakes
- Show the character's internal thoughts, fears, hopes, and dreams
- Use environmental details to reflect emotional states
- Create genuine dramatic tension and satisfying emotional resolution
- The feature should feel like a life-changing discovery, not just a tool
- Precise 6.5-second timing with perfect camera concealment

GENERATE 8 CONNECTED CHUNKS that tell the complete story of:
A devastated professional discovering "{feature_name}" and achieving life-changing transformation from crushing failure to triumphant success and future vision.

Each chunk must demonstrate different aspects of the feature while showing the same character's emotional and professional journey through different environments.

"""
```

```python
    payload = {
        "model": PERPLEXITY_MODEL,
        "messages": [
            {"role": "system", "content": "You are an expert cinematic storyteller who creates dramatic feature demonstration narratives that showcase real business va
            {"role": "user", "content": enhanced_story_prompt}
        ],
        "max_tokens": 6000,
        "temperature": 0.7
    }

    response = requests.post(url, headers=headers, json=payload)
    response.raise_for_status()
    return response.json()['choices'][0]['message']['content']


# ========================
# CHUNK PARSING FUNCTION (MISSING FUNCTION ADDED)
# ========================
def parse_feature_driven_chunks(chunks_output: str) -> Dict[str, Any]:
    """Parse feature-driven chunks with comprehensive validation and extraction."""

    chunk_pattern = r'CHUNK \d+:'
    parts = re.split(chunk_pattern, chunks_output)
    chunk_headers = re.findall(chunk_pattern, chunks_output)
    chunk_contents = parts[1:]

    if len(chunk_headers) == len(chunk_contents) and len(chunk_headers) == 8:
        parsed_chunks = []
        character_dna = ""
        feature_values = []
        emotional_beats = []
        pain_points_addressed = []
        feature_capabilities = []
        quantified_benefits = []
        use_case_contexts = []

        for i, (header, content) in enumerate(zip(chunk_headers, chunk_contents)):
            full_chunk = header + content.strip()
            parsed_chunks.append(full_chunk)

            # Extract character DNA from first chunk (should be identical across all)
            if i == 0:
                dna_match = re.search(r'CHARACTER DNA: (.+?)(?=\[|$)', full_chunk, re.DOTALL)
                if dna_match:
                    character_dna = dna_match.group(1).strip()

            # Extract feature-specific elements
            pain_point_match = re.search(r'\[PAIN POINT ADDRESSED\]: (.+?)(?=\n\[|\n*$)', full_chunk, re.DOTALL)
            if pain_point_match:
                pain_points_addressed.append(pain_point_match.group(1).strip())

            capability_match = re.search(r'\[FEATURE CAPABILITY\]: (.+?)(?=\n\[|\n*$)', full_chunk, re.DOTALL)
            if capability_match:
                feature_capabilities.append(capability_match.group(1).strip())

            benefit_match = re.search(r'\[QUANTIFIED BENEFIT\]: (.+?)(?=\n\[|\n*$)', full_chunk, re.DOTALL)
            if benefit_match:
                quantified_benefits.append(benefit_match.group(1).strip())

            emotional_match = re.search(r'\[EMOTIONAL BEAT\]: (.+?)(?=\n\[|\n*$)', full_chunk, re.DOTALL)
            if emotional_match:
                emotional_beats.append(emotional_match.group(1).strip())

            use_case_match = re.search(r'\[USE CASE CONTEXT\]: (.+?)(?=\n\[|\n*$)', full_chunk, re.DOTALL)
            if use_case_match:
                use_case_contexts.append(use_case_match.group(1).strip())

        # Validate feature integration
        feature_integration_score = 0
        if len(pain_points_addressed) >= 6:
            feature_integration_score += 25
        if len(feature_capabilities) >= 6:
            feature_integration_score += 25
        if len(quantified_benefits) >= 6:
            feature_integration_score += 25
        if len(use_case_contexts) >= 6:
            feature_integration_score += 25

        return {
            "parsed_chunks": parsed_chunks,
            "character_dna": character_dna,
            "pain_points_addressed": pain_points_addressed,
            "feature_capabilities": feature_capabilities,
            "quantified_benefits": quantified_benefits,
            "emotional_beats": emotional_beats,
            "use_case_contexts": use_case_contexts,
            "validation_passed": True,
            "chunk_count": len(parsed_chunks),
            "feature_integration_score": feature_integration_score,
            "validation_details": {
                "has_character_dna": bool(character_dna),
                "pain_points_count": len(pain_points_addressed),
                "capabilities_count": len(feature_capabilities),
                "benefits_count": len(quantified_benefits),
                "emotional_beats_count": len(emotional_beats),
                "use_cases_count": len(use_case_contexts)
            }
        }
    else:
        return {
            "parsed_chunks": [],
            "character_dna": "",
            "pain_points_addressed": [],
            "feature_capabilities": [],
            "quantified_benefits": [],
            "emotional_beats": [],
            "use_case_contexts": [],
            "validation_passed": False,
            "chunk_count": len(chunk_headers),
            "feature_integration_score": 0,
            "error": f"Expected 8 chunks, found {len(chunk_headers)}",
            "validation_details": {
                "has_character_dna": False,
                "pain_points_count": 0,
                "capabilities_count": 0,
                "benefits_count": 0,
                "emotional_beats_count": 0,
                "use_cases_count": 0
            }
        }

# ========================
# VEO3 TECHNICAL DOCUMENTATION ENGINE
# ========================
class VEO3DocumentationEngine:
    """Comprehensive VEO3 prompt engineering documentation system."""

    def __init__(self):
        self.documentation_data = {}

    def document_master_agent_prompt(self, feature_analysis: Dict, narrative_structure: Dict) -> str:
        """Generate and document the master agent prompt with feature integration."""

        master_prompt = f"""
MASTER VEO3 AGENT PROMPT - FEATURE-DRIVEN CINEMATIC STORYTELLING

==== FEATURE ANALYSIS FOUNDATION ====
Feature Name: {feature_analysis.get('feature_name', 'N/A')}
Feature Category: {feature_analysis.get('feature_category', 'N/A')}
Core Value Propositions: {json.dumps(feature_analysis.get('core_value_propositions', []), indent=2)}
Customer Pain Points: {json.dumps(feature_analysis.get('customer_pain_points', []), indent=2)}
Key Capabilities: {json.dumps(feature_analysis.get('key_capabilities', []), indent=2)}
Quantified Benefits: {json.dumps(feature_analysis.get('quantified_benefits', []), indent=2)}

==== STAKEHOLDER JOURNEY MAPPING ====
Primary User Persona: {feature_analysis.get('stakeholder_journeys', {}).get('primary_user', {}).get('persona', 'N/A')}
Current State: {feature_analysis.get('stakeholder_journeys', {}).get('primary_user', {}).get('current_state', 'N/A')}
Pain Points: {json.dumps(feature_analysis.get('stakeholder_journeys', {}).get('primary_user', {}).get('pain_points', []))}
Desired State: {feature_analysis.get('stakeholder_journeys', {}).get('primary_user', {}).get('desired_state', 'N/A')}
Success Indicators: {json.dumps(feature_analysis.get('stakeholder_journeys', {}).get('primary_user', {}).get('success_indicators', []))}
Emotional Arc: {feature_analysis.get('stakeholder_journeys', {}).get('primary_user', {}).get('emotional_arc', 'N/A')}

==== USE CASE INTEGRATION ====
{json.dumps(feature_analysis.get('use_cases', []), indent=2)}

==== NARRATIVE EXCELLENCE FRAMEWORK ====
Central Conflict: {feature_analysis.get('narrative_themes', {}).get('central_conflict', 'N/A')}
Resolution Path: {feature_analysis.get('narrative_themes', {}).get('resolution_path', 'N/A')}
Transformation Story: {feature_analysis.get('narrative_themes', {}).get('transformation_story', 'N/A')}
Emotional Beats: {json.dumps(feature_analysis.get('narrative_themes', {}).get('emotional_beats', []))}
Visual Metaphors: {json.dumps(feature_analysis.get('narrative_themes', {}).get('visual_metaphors', []))}
Story Settings: {json.dumps(feature_analysis.get('narrative_themes', {}).get('story_settings', []))}

==== 8-CHUNK EMOTIONAL PROGRESSION ====
{json.dumps(narrative_structure.get('emotional_progression', {}), indent=2)}

==== VEO3 TECHNICAL SPECIFICATIONS ====
1. PRECISE TIMING CONTROL:
   - Total Duration: EXACTLY 6.5 seconds per chunk
   - Camera Movement: 5.5 seconds lateral dolly movement
   - Concealment Phase: 1.0 seconds behind right object
   - NO footage after concealment point

2. CHARACTER CONSISTENCY PROTOCOL:
   - Identical character DNA across all 8 chunks
   - Same facial features, clothing, and physical characteristics
   - Consistent behavior patterns aligned with stakeholder persona
   - Professional appearance matching target user persona

3. SCENE DIVERSITY WITH PURPOSE:
   - Each chunk shows COMPLETELY different professional environment
   - Progressive feature demonstration through varied contexts
   - Visual variety supporting emotional progression and use case integration

4. CAMERA MOVEMENT SPECIFICATION:
   {EXACT_CAMERA_TEXT}

5. FEATURE DEMONSTRATION REQUIREMENTS:
   - Visual communication of feature value without on-screen text
   - Authentic professional scenarios from use case analysis
   - Clear capability demonstration in realistic contexts
   - Transformation elements showing before/after improvements

==== QUALITY ASSURANCE FRAMEWORK ====
- Feature value proposition alignment verification
- Stakeholder journey progression validation
- Use case integration authenticity check
- Emotional engagement escalation confirmation
- VEO3 technical specification compliance
- Character consistency across all chunks
- Professional scenario authenticity validation
- Quantified benefit visual representation assessment

==== SUCCESS CRITERIA ====
- Complete feature story told across 8 connected chunks
- Authentic stakeholder transformation demonstrated
- Real use cases integrated into narrative
- Professional quality cinematic presentation
- Precise timing with perfect camera concealment
- Identical characters throughout different professional scenarios
- Clear feature value communication without text overlays
"""

        # Save master prompt with feature context
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
        feature_name = feature_analysis.get('feature_name', 'feature').replace(' ', '_').lower()
        prompt_file = os.path.join(DOCUMENTATION_FOLDER, f"master_prompt_{feature_name}_{timestamp}.txt")
        with open(prompt_file, 'w', encoding='utf-8') as f:
            f.write(master_prompt)

        self.documentation_data['master_prompt'] = master_prompt
        self.documentation_data['feature_context'] = feature_analysis
        return master_prompt

    def document_veo3_generation_process(self, chunk_index: int, prompt_text: str,
                                       success: bool, error_msg: str = None) -> Dict:
        """Document each VEO3 generation attempt with success/failure analysis."""

        process_doc = {
            "timestamp": datetime.now().isoformat(),
            "chunk_index": chunk_index,
            "prompt_length": len(prompt_text),
            "prompt_preview": prompt_text[:500] + "..." if len(prompt_text) > 500 else prompt_text,
            "generation_success": success,
            "error_message": error_msg,
            "veo3_compatibility_notes": self._analyze_veo3_compatibility(prompt_text),
            "optimization_suggestions": self._generate_optimization_suggestions(prompt_text, success, error_msg)
        }

        # Save individual process documentation
        doc_file = os.path.join(DOCUMENTATION_FOLDER, f"veo3_process_chunk_{chunk_index}_{datetime.now().strftime('%Y%m%d_%H%M%S')}.json")
        with open(doc_file, 'w', encoding='utf-8') as f:
            json.dump(process_doc, f, indent=2)

        return process_doc
    def _analyze_veo3_compatibility(self, prompt_text: str) -> Dict[str, Any]:
        """Analyze prompt for VEO3 compatibility issues."""

        compatibility_analysis = {
            "character_consistency_elements": len(re.findall(r'CHARACTER DNA|same.*character|identical.*person', prompt_text, re.IGNORECASE)),
            "timing_specifications": len(re.findall(r'6\.5.*second|duration.*6\.5|exactly.*6\.5', prompt_text, re.IGNORECASE)),
            "camera_movement_details": len(re.findall(r'dolly|camera.*move|left.*right|behind.*object', prompt_text, re.IGNORECASE)),
            "technical_specifications": len(re.findall(r'16:9|aspect.*ratio|resolution|quality', prompt_text, re.IGNORECASE)),
            "negative_prompt_elements": len(re.findall(r'no.*text|avoid|never|must not', prompt_text, re.IGNORECASE)),
            "scene_description_depth": len(prompt_text.split()),
            "veo3_optimization_score": 0
        }

        # Calculate optimization score
        score = 0
        score += min(compatibility_analysis["character_consistency_elements"] * 10, 30)
        score += min(compatibility_analysis["timing_specifications"] * 15, 45)
        score += min(compatibility_analysis["camera_movement_details"] * 5, 25)

        compatibility_analysis["veo3_optimization_score"] = score

        return compatibility_analysis

    def _generate_optimization_suggestions(self, prompt_text: str, success: bool, error_msg: str) -> List[str]:
        """Generate suggestions for improving VEO3 prompts."""

        suggestions = []

        if not success and error_msg:
            if "timeout" in error_msg.lower():
                suggestions.append("Reduce prompt complexity - VEO3 may be timing out on overly detailed prompts")
            if "character" in error_msg.lower():
                suggestions.append("Simplify character descriptions while maintaining consistency")
            if "technical" in error_msg.lower():
                suggestions.append("Review technical specifications for VEO3 compatibility")

        # Analyze prompt for improvements
        if len(prompt_text) > 2000:
            suggestions.append("Consider reducing prompt length - VEO3 performs better with concise, focused prompts")

        if prompt_text.count("MUST") > 5:
            suggestions.append("Reduce aggressive language - use positive framing instead of negative constraints")

        if "character DNA" not in prompt_text.lower():
            suggestions.append("Add explicit character DNA section for better consistency")

        return suggestions

# =========================
# QUALITY ASSURANCE FRAMEWORK
# =========================
class QualityAssuranceFramework:
    """Comprehensive quality assessment beyond timing."""

    def __init__(self, feature_analysis: Dict, narrative_structure: Dict):
        self.feature_analysis = feature_analysis
        self.narrative_structure = narrative_structure

    def assess_narrative_coherence(self, chunks: List[str]) -> Dict[str, Any]:
        """Assess narrative flow and coherence across chunks."""

        coherence_assessment = {
            "story_progression_score": 0,
            "character_arc_consistency": 0,
            "thematic_coherence": 0,
            "feature_integration_score": 0,
            "scene_transitions": [],
            "narrative_gaps": [],
            "improvement_recommendations": []
        }

        # Analyze story progression
        story_elements = []
        for i, chunk in enumerate(chunks):
            setting_match = re.search(r'\[SETTING\]: (.+?)(?=\n\[|\n*$)', chunk, re.DOTALL)
            action_match = re.search(r'\[ACTION\]: (.+?)(?=\n\[|\n*$)', chunk, re.DOTALL)

            if setting_match and action_match:
                story_elements.append({
                    "chunk": i+1,
                    "setting": setting_match.group(1).strip(),
                    "action": action_match.group(1).strip()
                })

        # Score story progression (0-100)
        coherence_assessment["story_progression_score"] = min(len(story_elements) * 12.5, 100)

        # Assess thematic coherence with feature focus
        feature_name = self.feature_analysis.get('feature_name', '').lower()
        core_values = ' '.join(self.feature_analysis.get('core_value_propositions', [])).lower()

        feature_mentions = 0
        for chunk in chunks:
            chunk_lower = chunk.lower()
            if feature_name in chunk_lower:
                feature_mentions += 1
            for value_word in core_values.split()[:5]:  # Top 5 words from value props
                if len(value_word) > 3 and value_word in chunk_lower:
                    feature_mentions += 0.5

        coherence_assessment["feature_integration_score"] = min(feature_mentions * 10, 100)
        coherence_assessment["thematic_coherence"] = min(feature_mentions * 8, 100)

        return coherence_assessment

    def assess_emotional_engagement(self, chunks: List[str]) -> Dict[str, Any]:
        """Assess emotional impact and engagement potential."""

        emotional_words = {
            "positive": ["success", "achieve", "improve", "better", "easy", "efficient", "happy", "satisfied", "confident", "excited"],
            "negative": ["struggle", "difficult", "problem", "challenge", "frustration", "error", "fail", "stress", "overwhelmed"],
            "transition": ["discover", "realize", "learn", "understand", "change", "transform", "grow", "adopt", "implement"]
        }

        engagement_assessment = {
            "emotional_progression_score": 0,
            "emotional_word_distribution": {"positive": 0, "negative": 0, "transition": 0},
            "engagement_hooks": [],
            "emotional_peaks": [],
            "improvement_suggestions": []
        }
        # Analyze emotional word distribution across chunks
        for i, chunk in enumerate(chunks):
            chunk_lower = chunk.lower()
            chunk_emotions = {"positive": 0, "negative": 0, "transition": 0}

            for category, words in emotional_words.items():
                for word in words:
                    if word in chunk_lower:
                        engagement_assessment["emotional_word_distribution"][category] += 1
                        chunk_emotions[category] += 1

            # Identify emotional peaks
            total_emotion = sum(chunk_emotions.values())
            if total_emotion > 3:
                engagement_assessment["emotional_peaks"].append(f"Chunk {i+1}: High emotional content")

        # Score emotional progression
        total_words = sum(engagement_assessment["emotional_word_distribution"].values())
        engagement_assessment["emotional_progression_score"] = min(total_words * 5, 100)

        return engagement_assessment

    def assess_veo3_compatibility(self, chunks: List[str]) -> Dict[str, Any]:
        """Assess VEO3-specific compatibility and optimization."""

        compatibility_assessment = {
            "technical_compliance_score": 0,
            "camera_instruction_quality": 0,
            "character_consistency_potential": 0,
            "scene_diversity_score": 0,
            "feature_demonstration_clarity": 0,
            "veo3_optimization_recommendations": []
        }

        # Assess technical compliance
        technical_elements = 0
        for chunk in chunks:
            if "6.5 second" in chunk or "6.5-second" in chunk:
                technical_elements += 1
            if "dolly" in chunk.lower():
                technical_elements += 1
            if "behind" in chunk.lower() and "object" in chunk.lower():
                technical_elements += 1

        compatibility_assessment["technical_compliance_score"] = min(technical_elements * 4, 100)

        # Assess camera instruction quality
        camera_instructions = 0
        for chunk in chunks:
            if re.search(r'\[CAMERA\]:', chunk):
                camera_instructions += 1

        compatibility_assessment["camera_instruction_quality"] = min(camera_instructions * 12.5, 100)

        # Assess scene diversity
        settings = set()
        for chunk in chunks:
            setting_match = re.search(r'\[SETTING\]: (.+?)(?=\n\[|\n*$)', chunk, re.DOTALL)
            if setting_match:
                settings.add(setting_match.group(1).strip().lower()[:30])  # First 30 chars for comparison

        compatibility_assessment["scene_diversity_score"] = min(len(settings) * 12.5, 100)

        # Assess feature demonstration clarity
        feature_demo_elements = 0
        for chunk in chunks:
            if re.search(r'\[FEATURE CAPABILITY\]:', chunk):
                feature_demo_elements += 1
            if re.search(r'\[QUANTIFIED BENEFIT\]:', chunk):
                feature_demo_elements += 1
            if re.search(r'\[VISUAL FEATURE DEMONSTRATION\]:', chunk):
                feature_demo_elements += 1

        compatibility_assessment["feature_demonstration_clarity"] = min(feature_demo_elements * 4, 100)

        return compatibility_assessment

    def generate_comprehensive_qa_report(self, chunks: List[str], video_paths: Dict[int, str]) -> Dict[str, Any]:
            """Generate comprehensive quality assurance report."""

            narrative_assessment = self.assess_narrative_coherence(chunks)
            emotional_assessment = self.assess_emotional_engagement(chunks)
            compatibility_assessment = self.assess_veo3_compatibility(chunks)

            overall_score = (
                narrative_assessment["story_progression_score"] * 0.25 +
                narrative_assessment["feature_integration_score"] * 0.25 +
                emotional_assessment["emotional_progression_score"] * 0.25 +
                compatibility_assessment["technical_compliance_score"] * 0.25
            )

            comprehensive_report = {
                "timestamp": datetime.now().isoformat(),
                "overall_quality_score": round(overall_score, 2),
                "narrative_coherence": narrative_assessment,
                "emotional_engagement": emotional_assessment,
                "veo3_compatibility": compatibility_assessment,
                "video_generation_success": {
                    "completed_videos": len(video_paths),
                    "success_rate": (len(video_paths) / 8) * 100,
                    "failed_chunks": [i for i in range(1, 9) if i not in video_paths]
                },
                "feature_integration_analysis": {
                    "feature_name": self.feature_analysis.get('feature_name', 'N/A'),
                    "value_propositions_integrated": len(self.feature_analysis.get('core_value_propositions', [])),
                    "pain_points_addressed": len(self.feature_analysis.get('customer_pain_points', [])),
                    "use_cases_incorporated": len(self.feature_analysis.get('use_cases', []))
                },
                "improvement_recommendations": self._generate_improvement_recommendations(
                    narrative_assessment, emotional_assessment, compatibility_assessment
                )
            }

            # Save comprehensive report
            timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
            feature_name = self.feature_analysis.get('feature_name', 'feature').replace(' ', '_').lower()
            report_file = os.path.join(QA_REPORTS_FOLDER, f"comprehensive_qa_report_{feature_name}_{timestamp}.json")
            with open(report_file, 'w', encoding='utf-8') as f:
                json.dump(comprehensive_report, f, indent=2)

            return comprehensive_report
    def _generate_improvement_recommendations(self, narrative_assess: Dict, emotional_assess: Dict,
                                            compatibility_assess: Dict) -> List[str]:
        """Generate specific improvement recommendations."""

        recommendations = []

        if narrative_assess["story_progression_score"] < 80:
            recommendations.append("Strengthen story progression with clearer scene-to-scene connections")

        if narrative_assess["feature_integration_score"] < 80:
            recommendations.append("Enhance feature integration - ensure each chunk clearly demonstrates specific capabilities")

        if emotional_assess["emotional_progression_score"] < 80:
            recommendations.append("Enhance emotional engagement with stronger emotional language and character development")

        if compatibility_assess["technical_compliance_score"] < 90:
            recommendations.append("Improve VEO3 technical compliance with more precise timing and camera specifications")

        if compatibility_assess["feature_demonstration_clarity"] < 80:
            recommendations.append("Clarify feature demonstration elements - make benefits more visually apparent")

        return recommendations

# ========================
# VIDEO PROCESSING FUNCTIONS
# ========================
def trim_video_to_exact_timing(input_path, output_path, duration_seconds=8):
    """Trim video to exact duration to stop precisely when camera is behind object."""
    try:
        if os.path.exists(output_path):
            os.remove(output_path)
        command = [
            'ffmpeg', '-y',
            '-i', input_path,
            '-t', str(duration_seconds),
            '-c:v', 'libx264',
            '-c:a', 'aac',
            '-crf', '18',
            '-preset', 'medium',
            '-pix_fmt', 'yuv420p',
            output_path
        ]

        result = subprocess.run(command, capture_output=True, text=True)
        if result.returncode != 0:
            return None, f"Error trimming video: {result.stderr}"
        return output_path, None
    except Exception as e:
        return None, f"Error in trim_video_to_exact_timing: {str(e)}"


def batch_trim_all_videos():
    """Copy all generated video chunks without trimming."""
    copied_paths = {}
    errors = []

    for i in range(1, 9):
        input_path = os.path.join(VIDEO_OUTPUT_FOLDER, f"connected_chunk_{i}.mp4")
        output_path = os.path.join(TRIMMED_VIDEO_FOLDER, f"trimmed_chunk_{i}.mp4")

        if os.path.exists(input_path):
            try:
                import shutil
                shutil.copy2(input_path, output_path)
                copied_paths[i] = output_path
            except Exception as e:
                errors.append(f"Chunk {i}: Failed to copy - {str(e)}")

    return copied_paths, errors

def generate_enhanced_video_with_documentation(chunk_text, chunk_index, character_dna,
                                               narrative_structure, feature_analysis,
                                               documentation_engine):
    """Generate video with comprehensive documentation and voice-over integration."""

    # Get emotional and cinematic elements for this chunk
    emotional_beat = narrative_structure.get('emotional_progression', {}).get(chunk_index, {})

    # Extract scene elements
    setting_match = re.search(r'\[SETTING\]: (.+?)(?=\n\[|\n*$)', chunk_text, re.DOTALL)
    action_match = re.search(r'\[ACTION\]: (.+?)(?=\n\[|\n*$)', chunk_text, re.DOTALL)
    feature_capability_match = re.search(r'\[FEATURE CAPABILITY\]: (.+?)(?=\n\[|\n*$)', chunk_text, re.DOTALL)
    quantified_benefit_match = re.search(r'\[QUANTIFIED BENEFIT\]: (.+?)(?=\n\[|\n*$)', chunk_text, re.DOTALL)
    emotional_beat_match = re.search(r'\[EMOTIONAL BEAT\]: (.+?)(?=\n\[|\n*$)', chunk_text, re.DOTALL)

    setting = setting_match.group(1).strip() if setting_match else f"Professional environment #{chunk_index}"
    action = action_match.group(1).strip() if action_match else f"Feature demonstration activity #{chunk_index}"
    feature_capability = feature_capability_match.group(1).strip() if feature_capability_match else feature_analysis.get('key_capabilities', ['Feature capability'])[0]
    quantified_benefit = quantified_benefit_match.group(1).strip() if quantified_benefit_match else feature_analysis.get('quantified_benefits', ['Measurable improvemen
    emotional_state = emotional_beat_match.group(1).strip() if emotional_beat_match else emotional_beat.get('emotion', 'Neutral')

    # Generate voice-over script for this chunk
    voiceover_script = generate_voiceover_script(chunk_text, chunk_index, feature_analysis, emotional_beat)

    # Create enhanced VEO3 prompt with voice-over integration
    enhanced_veo3_prompt = f"""

FEATURE-DRIVEN CINEMATIC CHUNK {chunk_index}/8 - PRECISE 6.5-SECOND TIMING WITH VOICE-OVER

=== CHARACTER CONSISTENCY (EXACT DNA) ===
{character_dna}

=== FEATURE DEMONSTRATION CONTEXT ===
Core Feature Capability: {feature_capability}
Quantified Benefit: {quantified_benefit}
Emotional State: {emotional_state} (Intensity: {emotional_beat.get('intensity', 5)}/10)
Narrative Beat: {emotional_beat.get('beat', 'Story development')}

=== UNIQUE SCENE SPECIFICATION ===
Setting: {setting}
Action: {action}

=== VOICE-OVER AUDIO INTEGRATION ===
**Cinematic Narrator Script (6.5 seconds):**
"{voiceover_script}"

**Audio Requirements:**
- - Primary Audio: Cinematic narrator voice-over (provided script above)
- **CRITICAL**: Use identical narrator voice across ALL video chunks - consistent male documentary narrator (Morgan Freeman style)
- Secondary Audio: Natural ambient sounds appropriate for {setting}
- Secondary Audio: Natural ambient sounds appropriate for {setting}
- Background Audio: Optional realistic conversations/ambient dialogue (NOT characters addressing camera)
- Audio Style: Professional cinematic quality, story-driven narrative tone
- Narrator Voice: Authoritative yet empathetic cinematic storyteller (NOT product demo voice)
- Audio Timing: Voice-over must align perfectly with 6.5-second visual duration

**Prohibited Audio Elements:**
- No on-screen text captions or subtitles
- No characters directly addressing the audience
- No product demo-style narration
- No technical jargon in dialogue
- Voice-over should not feel like advertising copy

=== CRITICAL TIMING REQUIREMENTS ===
Total Duration: EXACTLY 6.5 seconds
Phase 1 (0-5.5s): {EXACT_CAMERA_TEXT}
Phase 2 (5.5-6.5s): Camera becomes fully concealed behind right-side object
MANDATORY: Video ENDS exactly when camera is completely hidden
Audio Sync: Voice-over script must complete within the 6.5-second timeframe

=== STAKEHOLDER JOURNEY ALIGNMENT ===
User Persona: {feature_analysis.get('stakeholder_journeys', {}).get('primary_user', {}).get('persona', 'Professional user')}
Journey Stage: {emotional_beat.get('stakeholder_stage', 'Feature exploration')}
Pain Point Addressed: {feature_analysis.get('customer_pain_points', ['User challenge'])[0] if feature_analysis.get('customer_pain_points') else 'User challenge'}

=== ADVANCED CINEMATIC REQUIREMENTS ===
- Emotional Progression: Show {emotional_state} building to {emotional_beat.get('intensity', 5)}/10 intensity
- Visual Storytelling: Use environment and actions to demonstrate feature value (complementing voice-over)
- Professional Quality: Cinematic lighting, composition, and movement
- Character Behavior: Natural, authentic interactions that never acknowledge camera
- Feature Integration: Clear demonstration of {feature_capability} within scene
- Audio-Visual Harmony: Visual action should complement voice-over narrative without literal description

=== TECHNICAL SPECIFICATIONS ===
- Resolution: Professional quality suitable for business presentation
- Aspect Ratio: 16:9 widescreen format
- Camera Movement: Precise dolly movement from left to right with exact timing
- Character Consistency: EXACT same people, faces, clothing, mannerisms as specified
- Scene Uniqueness: Completely different from all other 7 chunks
- Timing Precision: Must end exactly at 6.5 seconds when camera is concealed
- Audio Quality: Professional cinematic audio with clear voice-over narration

=== QUALITY CONTROLS ===
- NO on-screen text, captions, or subtitles
- Characters never look directly at camera
- NO footage continues after camera concealment
- Maintain consistent visual style across story
- Demonstrate clear feature value through action and environment
- Support emotional progression of overall narrative
- Voice-over must carry the story and emotional arc
- Audio complements visuals without describing them literally

PRIORITY: Create a professional, feature-demonstrating scene with integrated cinematic voice-over narration that ends with precise camera concealment at exactly 6.5 se

**Voice-Over Script to Include:**
"{voiceover_script}"

"""

    # Multi-model fallback with documentation
    model_endpoints = [
    "veo-3.0-generate-preview", # High quality production
    "veo-3.0-fast-generate-preview",    # Fast preview version
    "veo-3.0-fast-generate-001",  # Fastest & most reliable
    "veo-3.0-generate-001",       # Standard preview version
    "veo-2.0-generate-001"         # Fallback to VEO 2.0
    ]

    base_url = "https://generativelanguage.googleapis.com/v1beta/models"
    headers = {
        "x-goog-api-key": GEMINI_API_KEY,
        "Content-Type": "application/json"
    }

    for model in model_endpoints:
        url = f"{base_url}/{model}:predictLongRunning"
        json_payload = {
            "instances": [{"prompt": enhanced_veo3_prompt}],
            "parameters": {
                "aspectRatio": "16:9",
                "negativePrompt": "different narrator voices, voice changes between scenes, multiple narrators, inconsistent voice tone, character changes, inconsister
                "personGeneration": "allow_all"
            }
        }

        try:
            response = requests.post(url, headers=headers, json=json_payload)
            if response.status_code == 200:
                operation_name = response.json().get('name')

                # Document successful generation start with voice-over integration
                documentation_engine.document_veo3_generation_process(
                    chunk_index, enhanced_veo3_prompt, True
                )

                return operation_name, None
            else:
                print(f"Model {model} failed with status {response.status_code}: {response.text}")
                continue

        except Exception as e:
            print(f"Exception calling model {model}: {str(e)}")
            continue

    # Document failure
    documentation_engine.document_veo3_generation_process(
        chunk_index, enhanced_veo3_prompt, False, "All model endpoints failed"
    )

    return None, "All model endpoints failed - please try again later"

def poll_veo3_video_generation_with_enhanced_tracking(operation_name, chunk_index):
    """Poll VEO3 video generation with enhanced tracking and auto-trim."""
    poll_url = f"https://generativelanguage.googleapis.com/v1beta/{operation_name}"
    headers = {"x-goog-api-key": GEMINI_API_KEY}

    try:
        poll_response = requests.get(poll_url, headers=headers)
        if poll_response.status_code != 200:
            return None, f"API poll failed: {poll_response.status_code} - {poll_response.text}"

        poll_json = poll_response.json()

        if poll_json.get('done'):
            video_uri = extract_video_uri_from_response(poll_json)
            if video_uri:
                # Download original video
                original_path = os.path.join(VIDEO_OUTPUT_FOLDER, f"connected_chunk_{chunk_index}.mp4")
                trimmed_path = os.path.join(TRIMMED_VIDEO_FOLDER, f"trimmed_chunk_{chunk_index}.mp4")

                video_resp = requests.get(video_uri, headers={"x-goog-api-key": GEMINI_API_KEY}, stream=True)
                if video_resp.status_code == 200:
                    with open(original_path, 'wb') as f:
                        for chunk in video_resp.iter_content(chunk_size=8192):
                            f.write(chunk)

                    # Automatically trim to exact 6.5 seconds
                    trimmed_result, trim_error = trim_video_to_exact_timing(original_path, trimmed_path, 6.5)
                    if trimmed_result:
                        return trimmed_path, None
                    else:
                        return original_path, f"Trimming failed: {trim_error}"
                else:
                    return None, f"Failed to download video: {video_resp.status_code} - {video_resp.text}"
            else:
                resp = poll_json.get('response', {})
                available_keys = list(resp.keys()) if resp else list(poll_json.keys())
                return None, f"Video URI not found. Available response keys: {available_keys}"
        else:
            return "processing", None
    except Exception as e:
        return None, f"Error polling video: {str(e)}"

def extract_video_uri_from_response(poll_json):
    """Extract video URI from various response formats."""
    try:
        resp = poll_json.get('response', {})
        if 'generateVideoResponse' in resp:
            gvr = resp['generateVideoResponse']
            if 'generatedSamples' in gvr and len(gvr['generatedSamples']) > 0:
                sample = gvr['generatedSamples'][0]
                if 'video' in sample:
                    uri = sample['video'].get('uri') or sample['video'].get('gcsUri')
                    if uri:
                        return uri
        # Additional extraction patterns...
    except Exception as e:
        print(f"Error extracting video URI: {e}")
        return None
    return None

def validate_video_file(file_path, filename):
    """Validate video file before processing"""
    try:
        # Check file size
        if os.path.getsize(file_path) == 0:
            return False, f"File {filename} is empty"

        # Try to load video with error handling
        test_clip = VideoFileClip(file_path)

        # Check if reader is available
        if test_clip.reader is None:
            test_clip.close()
            return False, f"Cannot read {filename} - unsupported format or corrupted"

        # Check duration
        if test_clip.duration is None or test_clip.duration <= 0:
            test_clip.close()
            return False, f"Invalid duration for {filename}"

        # Try to get first frame to ensure video is readable
        try:
            test_clip.get_frame(0)
        except Exception as e:
            test_clip.close()
            return False, f"Cannot read frames from {filename}: {str(e)}"

        duration = test_clip.duration
        test_clip.close()
        return True, f"Valid video: {duration:.2f}s"

    except Exception as e:
        return False, f"Error loading {filename}: {str(e)}"

def generate_voiceover_script(chunk_text, chunk_index, feature_analysis, emotional_beat):
    """Generate cinematic narrator voice-over script with storytelling focus."""

    # Extract key elements from chunk
    setting_match = re.search(r'\[SETTING\]: (.+?)(?=\n\[|\n*$)', chunk_text, re.DOTALL)
    action_match = re.search(r'\[ACTION\]: (.+?)(?=\n\[|\n*$)', chunk_text, re.DOTALL)
    pain_point_match = re.search(r'\[PAIN POINT ADDRESSED\]: (.+?)(?=\n\[|\n*$)', chunk_text, re.DOTALL)
    feature_capability_match = re.search(r'\[FEATURE CAPABILITY\]: (.+?)(?=\n\[|\n*$)', chunk_text, re.DOTALL)
    benefit_match = re.search(r'\[QUANTIFIED BENEFIT\]: (.+?)(?=\n\[|\n*$)', chunk_text, re.DOTALL)

    setting = setting_match.group(1).strip() if setting_match else "Professional workspace"
    action = action_match.group(1).strip() if action_match else "Feature demonstration"
    pain_point = pain_point_match.group(1).strip() if pain_point_match else "User challenges"
    capability = feature_capability_match.group(1).strip() if feature_capability_match else "Feature capability"
    benefit = benefit_match.group(1).strip() if benefit_match else "Improved efficiency"

    # Get emotional context
    emotion = emotional_beat.get('emotion', 'Neutral')
    intensity = emotional_beat.get('intensity', 5)

    # Map to 8 emotional beats for cinematic progression
    emotional_beat_mapping = {
        1: "Frustration - Current struggle and pain",
        2: "Escalation - Problem becomes critical",
        3: "Discovery - Hope emerges with solution",
        4: "Hope - Initial optimism and curiosity",
        5: "Engagement - Active exploration and trial",
        6: "Confidence - Growing trust and adoption",
        7: "Transformation - Clear improvement and success",
        8: "Celebration - Achievement and mastery"
    }

    current_beat = emotional_beat_mapping.get(chunk_index, "Story development")

    voiceover_prompt = f"""

You are a master cinematic narrator scriptwriter creating voice-over narration for a feature story video.

CONTEXT:
- Chunk {chunk_index}/8 in cinematic feature story
- Feature: {feature_analysis.get('feature_name', 'Product Feature')}
- Setting: {setting}
- Action: {action}
- Pain Point: {pain_point}
- Capability: {capability}
- Benefit: {benefit}
- Emotional Beat: {current_beat}
- Emotion: {emotion} (Intensity: {intensity}/10)

CINEMATIC VOICE-OVER REQUIREMENTS:

**Purpose**: Carry the story since no on-screen text is allowed
**Style**: Cinematic narrator voice (NOT product demo or character speaking to camera)
**Duration**: EXACTLY 6.5 seconds when spoken at natural cinematic pace
**Tone**: {emotion} with {intensity}/10 emotional intensity - storytelling, not selling

**8-Beat Emotional Progression Guidelines**:
- Chunk 1-2 (Frustration/Escalation): Show struggle, mounting pressure, current pain
- Chunk 3-4 (Discovery/Hope): Introduce possibility, emerging solution, growing optimism
- Chunk 5-6 (Engagement/Confidence): Show active use, building trust, visible improvement
- Chunk 7-8 (Transformation/Celebration): Demonstrate success, achievement, mastery

**Narrative Integration Rules**:
- Must align with the emotional arc and highlight: pain points в†’ discovery в†’ transformation в†’ success
- Reinforce feature benefits in plain, cinematic storytelling language (not technical jargon)
- Feel like a movie narrator, not a product demonstration
- Use story-driven language that complements visual action
- Should enhance what's shown visually without describing it literally

**Voice-Over Content Focus for Chunk {chunk_index}**:
Current Beat: {current_beat}
Story Element: {pain_point if chunk_index <= 2 else capability if chunk_index <= 6 else benefit}

**Cinematic Language Examples**:
- Instead of "The feature provides instant feedback" в†’ "In that moment, clarity emerged"
- Instead of "Users can track sentiment" в†’ "Understanding flowed like never before"
- Instead of "55% increase in engagement" в†’ "Connection sparked, then ignited"
- Instead of "The platform automates processes" в†’ "What once felt impossible became effortless"

Generate ONLY the voice-over script text (no directions, no timestamps, just the cinematic narration).
The script should be exactly the right length for 6.5 seconds when read at natural storytelling pace.

Example length reference: "When deadlines crush your spirit and manual work steals your dreams, you search for something more."

"""

    url = "https://api.perplexity.ai/chat/completions"
    headers = {
        "Authorization": f"Bearer {PERPLEXITY_API_KEY}",
        "Content-Type": "application/json"
    }

    payload = {
        "model": PERPLEXITY_MODEL,
        "messages": [
            {"role": "system", "content": "You are an expert cinematic narrator scriptwriter who creates emotionally engaging story-driven voice-over scripts for featu
            {"role": "user", "content": voiceover_prompt}
        ],
        "max_tokens": 150,
        "temperature": 0.7
    }

    try:
        response = requests.post(url, headers=headers, json=payload)
        response.raise_for_status()
        script = response.json()['choices'][0]['message']['content'].strip()

        # Clean up the script
        script = re.sub(r'^["\']*', '', script)  # Remove leading quotes
        script = re.sub(r'["\']*$', '', script)  # Remove trailing quotes
        script = script.replace('\n', ' ').strip()

        return script

    except Exception as e:
        # Enhanced fallback scripts with cinematic storytelling focus
        cinematic_fallback_scripts = {
            1: "Every day, the weight of inefficiency grows heavier, dreams deferred by endless manual struggles.",
            2: "Pressure builds as precious moments slip away, lost to systems that seem designed to fail.",
            3: "But in the darkness of frustration, a spark of possibility begins to shine.",
            4: "Hope emerges as innovation meets necessity, promising a different tomorrow.",
            5: "With careful steps forward, transformation begins to take shape before their eyes.",
            6: "Confidence grows as new capabilities unlock potential long thought impossible.",
            7: "The metamorphosis is completeвЂ"efficiency flows where chaos once reigned supreme.",
            8: "Success becomes the new reality, a testament to the power of perfect solutions."
        }

        return cinematic_fallback_scripts.get(chunk_index, "In this moment, change becomes possible through innovation and determination.")


def create_feature_demonstration_final_video():
    """Enhanced final video creation using MoviePy with validation."""
    try:
        # Collect all video files (try trimmed first, then original)
        video_files = []
        for i in range(1, 9):
            # First try trimmed videos
            trimmed_path = os.path.join(TRIMMED_VIDEO_FOLDER, f"trimmed_chunk_{i}.mp4")
            original_path = os.path.join(VIDEO_OUTPUT_FOLDER, f"connected_chunk_{i}.mp4")

            video_path = None
            if os.path.exists(trimmed_path):
                video_path = trimmed_path
            elif os.path.exists(original_path):
                video_path = original_path
                st.info(f"Using original video for chunk {i}")

            if video_path:
                # Validate each video file
                is_valid, message = validate_video_file(video_path, f"chunk_{i}.mp4")
                if is_valid:
                    video_files.append(video_path)
                    st.info(f"вњ… Chunk {i}: {message}")
                else:
                    st.warning(f"вљ пёЏ Chunk {i}: {message}")
            else:
                st.warning(f"вљ пёЏ Missing video for chunk {i}")

        if len(video_files) == 0:
            return None, "No valid videos found for combination"

        # Create temporary directory for processing
        temp_dir = tempfile.mkdtemp()
        processed_clips = []

        try:
            # Process each video
```

```python
            st.info(f"рџЋ¬ Processing {len(video_files)} videos for combination...")

            for i, video_path in enumerate(video_files):
                try:
                    # Load video clip
                    clip = VideoFileClip(video_path)

                    # Keep full duration (no additional trimming)
                    cut_clip = clip.copy()

                    processed_clips.append(cut_clip)

                except Exception as e:
                    st.error(f"Error processing video {i+1}: {str(e)}")
                    # Clean up any clips we've processed so far
                    for processed_clip in processed_clips:
                        try:
                            processed_clip.close()
                        except:
                            pass
                    return None, f"Failed to process video {i+1}: {str(e)}"

            # Combine all clips
            st.info("рџ”„ Combining all video clips...")
            final_video = concatenate_videoclips(processed_clips, method="compose")

            # Save final video
            output_path = "feature_demonstration_final_enhanced.mp4"
            temp_audio_path = os.path.join(temp_dir, 'temp-audio.m4a')

            final_video.write_videofile(
                output_path,
                codec='libx264',
                audio_codec='aac',
                verbose=False,
                logger=None,
                temp_audiofile=temp_audio_path,
                remove_temp=True
            )

            # Clean up clips
            for clip in processed_clips:
                clip.close()
            final_video.close()

            # Clean up temp directory
            shutil.rmtree(temp_dir, ignore_errors=True)

            return output_path, None

        except Exception as e:
            # Enhanced cleanup
            for clip in processed_clips:
                try:
                    clip.close()
                except:
                    pass

            shutil.rmtree(temp_dir, ignore_errors=True)
            return None, f"Error combining videos: {str(e)}"

    except Exception as e:
        return None, f"Error in final video creation: {str(e)}"


# =========================
# STREAMLIT UI - ENHANCED WITH PROPER FEATURE ANALYSIS
# =========================
st.set_page_config(page_title="AI-Powered Cinematic Video Script Generator", page_icon="рџЋ¬", layout="wide")
st.title("рџЋ¬ AI-Powered Cinematic Video Script Generator")
st.write("**Professional Feature Analysis в†’ Cinematic Storytelling Pipeline**")

# Status indicators
col1, col2, col3, col4 = st.columns(4)
with col1:
    st.success("рџ”Ќ Feature Analysis Engine: вњ…")
with col2:
    st.success("рџ“– Narrative Excellence: вњ…")
with col3:
    st.success("рџ“‹ VEO3 Documentation: вњ…")
with col4:
    st.success("рџЋЇ Quality Assurance: вњ…")

# Initialize session state
if 'feature_analysis' not in st.session_state:
    st.session_state.feature_analysis = {}
if 'feature_validation' not in st.session_state:
    st.session_state.feature_validation = {}
if 'narrative_structure' not in st.session_state:
    st.session_state.narrative_structure = {}
if 'parsed_chunk_data' not in st.session_state:
    st.session_state.parsed_chunk_data = {}
if 'documentation_engine' not in st.session_state:
    st.session_state.documentation_engine = VEO3DocumentationEngine()
if 'qa_framework' not in st.session_state:
    st.session_state.qa_framework = None
if 'master_prompt' not in st.session_state:
    st.session_state.master_prompt = ""
if 'operations' not in st.session_state:
    st.session_state.operations = {}
if 'video_paths' not in st.session_state:
    st.session_state.video_paths = {}
if 'final_qa_report' not in st.session_state:
    st.session_state.final_qa_report = {}
if 'voiceover_scripts' not in st.session_state:
    st.session_state.voiceover_scripts = {}


# Step 1: Feature Analysis Engine (ENHANCED)
st.header("рџ”Ќ Step 1: Feature Analysis Engine")
st.write("**Process detailed feature descriptions to extract value propositions, pain points, and stakeholder journeys**")

# Feature description input with example
st.subheader("Feature Description Input")
st.write("Provide a comprehensive feature description including capabilities, benefits, and use cases:")

# Example feature for reference
with st.expander("рџ“‹ Example Feature Description Format"):
    example_feature = """Feature Name: InsightSync AI-Powered Meeting Intelligence

Capabilities:
-Automatically records, transcribes, and summarizes live meetings across Zoom, Teams, and Google Meet
-AI-driven key-point extraction with action-item identification and responsibility assignment
```

```
-Multi-language translation for global team participation
-Integration with project management tools (Jira, Trello, Asana) for seamless task syncing
-Smart search functionality enabling retrieval of past decisions, quotes, or discussions instantly

Benefits:
-Saves up to 40 percent of meeting follow-up time by auto-generating summaries and task lists
-Increases accountability by ensuring every decision is captured and assigned
-Enhances collaboration for distributed teams with real-time translation
-Improves knowledge retention by archiving and indexing all discussions
-Reduces duplicate meetings by providing searchable historical context

Use Cases:
Enterprise Teams: Automatically create sprint backlogs from sprint planning meetings
Consulting Firms: Capture client requirements accurately and sync them to project trackers
Universities: Provide real-time translated lecture transcripts for international students
Healthcare: Document multi-disciplinary case discussions for compliance and follow-up
Startups: Enable lean teams to avoid redundancy by retrieving key past decisions on demand"""

    st.code(example_feature)

# Main feature input
# Main feature input
feature_description = st.text_area(
    "Enter your detailed feature description:",
    height=400,
    placeholder="Include: Feature name, capabilities, benefits (with numbers if available), and specific use cases...",
    help="Provide comprehensive details about your feature including what it does, how it helps users, and specific scenarios where it's used."
)

# Feature validation
if feature_description:
    analysis_engine = EnhancedFeatureAnalysisEngine()
    validation = analysis_engine.validate_feature_description(feature_description)
    st.session_state.feature_validation = validation

    if validation['completeness_score'] >= 75:
        st.success(f"вњ… Feature description quality: {validation['completeness_score']}/100 - Excellent!")
    elif validation['completeness_score'] >= 50:
        st.warning(f"вљ пёЏ Feature description quality: {validation['completeness_score']}/100 - Good, but could be enhanced")
    else:
        st.error(f"вќЊ Feature description quality: {validation['completeness_score']}/100 - Needs more detail")

    if validation['missing_elements']:
        st.write("**Missing elements:**")
        for element in validation['missing_elements']:
            st.write(f"вЂў {element.title()}")

    if validation['recommendations']:
        st.write("**Recommendations:**")
        for rec in validation['recommendations']:
            st.write(f"вЂў {rec}")

# Feature analysis button
if st.button("рџ”Ќ Analyze Feature", type="primary", disabled=not feature_description):
    if feature_description:
        with st.spinner("рџ”Ќ Analyzing feature description with AI-powered extraction..."):
            analysis_engine = EnhancedFeatureAnalysisEngine()
            st.session_state.feature_analysis = analysis_engine.parse_feature_description(feature_description)

            if st.session_state.feature_analysis:
                st.success("вњ… Feature analysis completed!")

                # Display comprehensive analysis results
                with st.expander("рџ“Љ Complete Feature Analysis Results", expanded=True):

                    col1, col2 = st.columns(2)

                    with col1:
                        st.write("**рџЋЇ Feature Overview**")
                        st.info(f"**Name:** {st.session_state.feature_analysis.get('feature_name', 'N/A')}")
                        st.info(f"**Category:** {st.session_state.feature_analysis.get('feature_category', 'N/A')}")

                        st.write("**рџ’Ў Core Value Propositions:**")
                        for i, value in enumerate(st.session_state.feature_analysis.get('core_value_propositions', []), 1):
                            st.write(f"{i}. {value}")

                        st.write("**вљЎ Key Capabilities:**")
                        for cap in st.session_state.feature_analysis.get('key_capabilities', []):
                            st.write(f"вЂў {cap}")

                    with col2:
                        st.write("**рџ¤ Customer Pain Points:**")
                        for pain in st.session_state.feature_analysis.get('customer_pain_points', []):
                            st.write(f"вЂў {pain}")

                        st.write("**рџ“€ Quantified Benefits:**")
                        for benefit in st.session_state.feature_analysis.get('quantified_benefits', []):
                            st.write(f"вЂў {benefit}")

                    # Stakeholder journey
                    st.write("**рџ‘Ґ Primary User Journey:**")
                    journey = st.session_state.feature_analysis.get('stakeholder_journeys', {}).get('primary_user', {})
                    if journey:
                        journey_col1, journey_col2 = st.columns(2)
                        with journey_col1:
                            st.write(f"**Persona:** {journey.get('persona', 'N/A')}")
                            st.write(f"**Current State:** {journey.get('current_state', 'N/A')}")
                            st.write(f"**Emotional Arc:** {journey.get('emotional_arc', 'N/A')}")
                        with journey_col2:
                            st.write(f"**Desired State:** {journey.get('desired_state', 'N/A')}")
                            if journey.get('success_indicators'):
                                st.write("**Success Indicators:**")
                                for indicator in journey.get('success_indicators', []):
                                    st.write(f"вЂў {indicator}")

                    # Use cases
                    st.write("**рџЋЇ Use Cases:**")
                    for i, case in enumerate(st.session_state.feature_analysis.get('use_cases', []), 1):
                        st.write(f"**{i}. {case.get('scenario', 'N/A')}**")
                        st.write(f"   Context: {case.get('context', 'N/A')}")
                        st.write(f"   Goal: {case.get('user_goal', 'N/A')}")
                        st.write(f"   Outcome: {case.get('outcome', 'N/A')}")

                    # Business impact
                    st.write("**рџ’ј Business Impact:**")
                    impact = st.session_state.feature_analysis.get('business_impact', {})
                    if impact:
                        for key, value in impact.items():
                            if isinstance(value, list):
                                st.write(f"**{key.replace('_', ' ').title()}:**")
                                for item in value:
                                    st.write(f"вЂў {item}")
                            else:
```

```python
                                st.write(f"**{key.replace('_', ' ').title()}:** {value}")
    else:
        st.error("Please enter a detailed feature description!")

# Step 2: Narrative Structure Generation
if st.session_state.feature_analysis:
    st.header("рџ“– Step 2: Narrative Excellence & Emotional Arc")
    st.write("**Generate cinematic storytelling structure based on feature analysis**")

    if st.button("рџ“– Generate Narrative Structure", type="primary"):
        with st.spinner("рџ“– Creating emotional arc and cinematic structure..."):
            narrative_engine = NarrativeExcellenceEngine(st.session_state.feature_analysis)
            st.session_state.narrative_structure = narrative_engine.generate_emotional_arc_structure()

            # Initialize QA Framework
            st.session_state.qa_framework = QualityAssuranceFramework(
                st.session_state.feature_analysis,
                st.session_state.narrative_structure
            )

            st.success("вњ… Narrative structure created!")

            # Display narrative structure
            with st.expander("рџ“– Emotional Arc & Story Structure", expanded=True):
                progression = st.session_state.narrative_structure.get('emotional_progression', {})

                # Create progression table
                progress_data = []
                for chunk_num, details in progression.items():
                    progress_data.append({
                        'Chunk': f"Scene {chunk_num}",
                        'Narrative Beat': details.get('beat', 'N/A'),
                        'Emotion': details.get('emotion', 'N/A'),
                        'Intensity': details.get('intensity', 5),
                        'Feature Focus': details.get('feature_focus', 'N/A'),
                        'Journey Stage': details.get('stakeholder_stage', 'N/A')
                    })

                if progress_data:
                    df = pd.DataFrame(progress_data)
                    st.dataframe(df, use_container_width=True)

                    # Emotional intensity chart
                    st.write("**Emotional Intensity Progression:**")
                    st.line_chart(df.set_index('Chunk')['Intensity'])

                # Show integrated elements
                st.write("**рџЋ Narrative Themes Integration:**")
                themes = st.session_state.narrative_structure.get('narrative_themes', {})
                st.write(f"**Central Conflict:** {themes.get('central_conflict', 'N/A')}")
                st.write(f"**Resolution Path:** {themes.get('resolution_path', 'N/A')}")
                st.write(f"**Visual Metaphors:** {', '.join(themes.get('visual_metaphors', []))}")

# Step 3: Generate Feature-Driven Story
if st.session_state.narrative_structure:
    st.header("рџЋ¬ Step 3: Generate Feature-Driven Cinematic Story")
    st.write("**Create story chunks with master agent prompt documentation**")

    if st.button("рџЋ¬ Generate Story & Documentation", type="primary"):
        with st.spinner("рџЋ¬ Generating comprehensive feature-driven story..."):

            # Generate and document master prompt
            st.session_state.master_prompt = st.session_state.documentation_engine.document_master_agent_prompt(
                st.session_state.feature_analysis,
                st.session_state.narrative_structure
            )

            # Generate feature-driven story
            story_output = generate_feature_driven_story_chunks(
                st.session_state.feature_analysis,
                st.session_state.narrative_structure
            )

            # Parse and validate story
            st.session_state.parsed_chunk_data = parse_feature_driven_chunks(story_output)

            if st.session_state.parsed_chunk_data.get('validation_passed'):
                st.success("вњ… Feature-driven cinematic story generated!")
                    # Generate voiceover scripts for each chunk
                with st.spinner("рџЋ™пёЏ Generating synchronized voiceover scripts..."):
                    parsed_chunks = st.session_state.parsed_chunk_data.get('parsed_chunks', [])
                    emotional_progression = st.session_state.narrative_structure.get('emotional_progression', {})

                    for i, chunk in enumerate(parsed_chunks, 1):
                        emotional_beat = emotional_progression.get(i, {})
                        voiceover_script = generate_voiceover_script(
                            chunk, i, st.session_state.feature_analysis, emotional_beat
                        )
                        st.session_state.voiceover_scripts[i] = voiceover_script

                st.success("рџЋ™пёЏ Voiceover scripts generated with emotional resonance!")



                # Success metrics
                col1, col2, col3, col4 = st.columns(4)
                with col1:
                    st.metric("Story Chunks", f"{st.session_state.parsed_chunk_data.get('chunk_count', 0)}/8")
                with col2:
                    st.metric("Character DNA", "вњ… Consistent" if st.session_state.parsed_chunk_data.get('character_dna') else "вќЊ Missing")
                with col3:
                    st.metric("Feature Integration", f"{st.session_state.parsed_chunk_data.get('feature_integration_score', 0)}/100")
                with col4:
                    st.metric("Pain Points", len(st.session_state.parsed_chunk_data.get('pain_points_addressed', [])))

                # Display comprehensive chunk analysis
                with st.expander("рџ”Ќ Feature Integration Analysis", expanded=False):
                    validation_details = st.session_state.parsed_chunk_data.get('validation_details', {})

                    col1, col2, col3 = st.columns(3)
                    with col1:
                        st.write("**Character Consistency:**")
                        st.write(f"вњ… Has DNA: {validation_details.get('has_character_dna', False)}")

                        st.write("**Pain Points:**")
                        st.write(f"Count: {validation_details.get('pain_points_count', 0)}/8")
                        for i, pain in enumerate(st.session_state.parsed_chunk_data.get('pain_points_addressed', [])[:3], 1):
                            st.write(f"{i}. {pain[:50]}...")

                    with col2:
                        st.write("**Feature Capabilities:**")
                        st.write(f"Count: {validation_details.get('capabilities_count', 0)}/8")
                        for i, cap in enumerate(st.session_state.parsed_chunk_data.get('feature_capabilities', [])[:3], 1):
                            st.write(f"{i}. {cap[:50]}...")

                        st.write("**Quantified Benefits:**")
                        st.write(f"Count: {validation_details.get('benefits_count', 0)}/8")

                    with col3:
                        st.write("**Emotional Beats:**")
                        st.write(f"Count: {validation_details.get('emotional_beats_count', 0)}/8")

                        st.write("**Use Case Integration:**")
                        st.write(f"Count: {validation_details.get('use_cases_count', 0)}/8")

                # Display master prompt documentation
                with st.expander("рџ“‹ Master Agent Prompt (Step-by-Step Documentation)", expanded=False):
                    st.write("**This is the complete master prompt used for story generation:**")
                    st.text_area("Master Agent Prompt:", st.session_state.master_prompt, height=500, key="master_prompt_display")

                    if st.button("рџ’s Download Master Prompt"):
                        timestamp = datetime.now().strftime('%Y%m%d_%H%M%S')
                        feature_name = st.session_state.feature_analysis.get('feature_name', 'feature').replace(' ', '_').lower()
                        st.download_button(
                            "рџ“„ Download Master Prompt",
                            st.session_state.master_prompt,
                            file_name=f"master_prompt_{feature_name}_{timestamp}.txt",
                            mime="text/plain"
                        )
                    # Download voiceover scripts
                    if st.session_state.voiceover_scripts:
                        if st.button("рџЋ™пёЏ Download Voiceover Scripts"):
                            timestamp = datetime.now().strftime('%Y%m%d_%H%M%S')
                            feature_name = st.session_state.feature_analysis.get('feature_name', 'feature').replace(' ', '_').lower()

                            # Create comprehensive voiceover document
                            voiceover_content = f"VOICEOVER SCRIPTS - {st.session_state.feature_analysis.get('feature_name', 'Feature Demo')}\n"
                            voiceover_content += f"Generated: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}\n"
                            voiceover_content += "=" * 50 + "\n\n"

                            for i in range(1, 9):
                                if i in st.session_state.voiceover_scripts:
                                    script = st.session_state.voiceover_scripts[i]
                                    word_count = len(script.split())
                                    duration = word_count / 2.5

                                    voiceover_content += f"CHUNK {i} VOICEOVER:\n"
                                    voiceover_content += f"Duration: {duration:.1f}s | Words: {word_count}\n"
                                    voiceover_content += f'Script: "{script}"\n\n'

                            st.download_button(
                                "рџЋ™пёЏ Download Complete Voiceover Scripts",
                                voiceover_content,
                                file_name=f"voiceover_scripts_{feature_name}_{timestamp}.txt",
                                mime="text/plain"
                            )

                # Display generated story (unaltered)
                with st.expander("рџ“– Generated Story Script (Unaltered Output)", expanded=False):
                    st.write("**Raw, unaltered output from the story generation:**")
                    st.text_area("Complete Story Script:", story_output, height=700, key="story_output_display")

                    if st.button("рџ’s Download Story Script"):
                        timestamp = datetime.now().strftime('%Y%m%d_%H%M%S')
                        feature_name = st.session_state.feature_analysis.get('feature_name', 'feature').replace(' ', '_').lower()
                        st.download_button(
                            "рџ“„ Download Story Script",
                            story_output,
                            file_name=f"story_script_{feature_name}_{timestamp}.txt",
                            mime="text/plain"
                        )
            else:
                st.error(f"вќЊ Story generation validation failed: {st.session_state.parsed_chunk_data.get('error', 'Unknown error')}")
                st.text_area("Debug - Raw Story Output:", story_output, height=400)

# Step 4: Video Generation with Documentation
if st.session_state.parsed_chunk_data.get('validation_passed'):
    st.header("рџЋҐ Step 4: Video Generation with VEO3 Documentation")
    st.write("**Generate videos with comprehensive process documentation**")

    parsed_chunks = st.session_state.parsed_chunk_data.get('parsed_chunks', [])
    character_dna = st.session_state.parsed_chunk_data.get('character_dna', '')

    # Display character DNA
    if character_dna:
        with st.expander("рџ§¬ Character DNA (Identical Across All Scenes)", expanded=False):
            st.info("This character description will be used consistently across all 8 video chunks:")
            st.text_area("Character DNA:", character_dna, height=200, key="character_dna_display")

    # Video generation for each chunk
    for i, chunk in enumerate(parsed_chunks, 1):
        with st.expander(f"рџЋ¬ Scene {i} - Feature Demonstration Video"):
            col1, col2 = st.columns([2, 1])

            with col1:
                st.text_area(f"Scene {i} Details:", chunk, height=300, key=f"chunk_display_{i}")

                # Show feature elements extracted from this chunk
                pain_points = st.session_state.parsed_chunk_data.get('pain_points_addressed', [])
                capabilities = st.session_state.parsed_chunk_data.get('feature_capabilities', [])
                benefits = st.session_state.parsed_chunk_data.get('quantified_benefits', [])

                if i-1 < len(pain_points):
                    st.success(f"**Pain Point:** {pain_points[i-1]}")
                if i-1 < len(capabilities):
                    st.info(f"**Feature Capability:** {capabilities[i-1]}")
                if i-1 < len(benefits):
                    st.warning(f"**Quantified Benefit:** {benefits[i-1]}")
                # Display voiceover script
                if i in st.session_state.voiceover_scripts:
                    st.markdown("**рџЋ™пёЏ Synchronized Voiceover Script:**")
                    st.markdown(f'*"{st.session_state.voiceover_scripts[i]}"*')

                    # Show script statistics
                    script = st.session_state.voiceover_scripts[i]
                    word_count = len(script.split())
                    estimated_duration = word_count / 2.5  # Average speaking pace
                    st.caption(f"рџ“Љ Words: {word_count} | Estimated duration: {estimated_duration:.1f}s | Target: 6.5s")


            with col2:
                if st.button(f"рџЋ¬ Generate Video {i}", key=f"gen_video_{i}"):
                    with st.spinner(f"рџЋ¬ Generating video {i} with feature integration..."):
                        operation, error = generate_enhanced_video_with_documentation(
                            chunk, i, character_dna,
                            st.session_state.narrative_structure,
                            st.session_state.feature_analysis,
                            st.session_state.documentation_engine
                        )

                        if operation:
                            st.session_state.operations[i] = operation
                            st.success(f"вњ… Video {i} generation started!")
                        else:
                            st.error(f"вќЊ Error: {error}")

                if i in st.session_state.operations:
                    if st.button(f"рџ”Ќ Check Status {i}", key=f"check_{i}"):
                        result, error = poll_veo3_video_generation_with_enhanced_tracking(
                            st.session_state.operations[i], i
                        )

                        if result == "processing":
                            st.info(f"рџ”„ Video {i} processing...")
                        elif result and result.endswith('.mp4'):
                            st.session_state.video_paths[i] = result
                            st.success(f"вњ… Video {i} completed - Precisely 6.5s!")
                        elif error:
                            st.error(f"вќЊ Error: {error}")

                if i in st.session_state.video_paths:
                    st.success(f"вњ… Video {i} ready - Perfect timing")
                    if os.path.exists(st.session_state.video_paths[i]):
                        st.video(st.session_state.video_paths[i])

                        # Show video file info
                        file_size = os.path.getsize(st.session_state.video_paths[i]) / (1024*1024)  # MB
                        st.caption(f"File size: {file_size:.1f} MB | Duration: 6.5s | Status: Trimmed")

# Step 5: Batch Operations
if len(st.session_state.operations) > 0:
    st.header("рџ”„ Step 5: Batch Operations")
    st.write("**Batch video generation and processing**")

    col1, col2, col3 = st.columns(3)

    with col1:
        if st.button("рџЋ¬ Generate All Remaining Videos"):
            remaining_chunks = [i for i in range(1, 9) if i not in st.session_state.operations]

            if remaining_chunks:
                with st.spinner(f"Generating {len(remaining_chunks)} remaining videos..."):
                    for chunk_idx in remaining_chunks:
                        if chunk_idx <= len(parsed_chunks):
                            operation, error = generate_enhanced_video_with_documentation(
                                parsed_chunks[chunk_idx-1], chunk_idx, character_dna,
                                st.session_state.narrative_structure,
                                st.session_state.feature_analysis,
                                st.session_state.documentation_engine
                            )

                            if operation:
                                st.session_state.operations[chunk_idx] = operation

                    st.success(f"вњ… Started generation for {len(remaining_chunks)} videos!")
            else:
                st.info("All videos already started!")

    with col2:
        if st.button("рџ”Ќ Check All Statuses"):
            with st.spinner("Checking all video generation statuses..."):
                status_updates = []

                for chunk_idx, operation in st.session_state.operations.items():
                    if chunk_idx not in st.session_state.video_paths:
                        result, error = poll_veo3_video_generation_with_enhanced_tracking(operation, chunk_idx)

                        if result == "processing":
                            status_updates.append(f"Video {chunk_idx}: Still processing...")
                        elif result and result.endswith('.mp4'):
                            st.session_state.video_paths[chunk_idx] = result
                            status_updates.append(f"Video {chunk_idx}: вњ… Completed!")
                        elif error:
                            status_updates.append(f"Video {chunk_idx}: вќЊ Error - {error}")

                if status_updates:
                    for update in status_updates:
                        if "вњ…" in update:
                            st.success(update)
                        elif "вќЊ" in update:
                            st.error(update)
                        else:
                            st.info(update)
                else:
                    st.info("No status updates - all videos either completed or not started")

    with col3:
        if len(st.session_state.video_paths) > 0:
            if st.button("рџ”§ Batch Trim All Videos"):
                with st.spinner("Trimming all videos to precise 6.5-second timing..."):
                    trimmed_paths, errors = batch_trim_all_videos()

                    if trimmed_paths:
                        st.success(f"вњ… Successfully trimmed {len(trimmed_paths)} videos!")
                        for i, path in trimmed_paths.items():
                            st.session_state.video_paths[i] = path

                    if errors:
                        st.warning("Some videos had trimming issues:")
                        for error in errors:
                            st.warning(error)

# Step 6: Quality Assurance & Final Video
if len(st.session_state.video_paths) >= 1:
    st.header("рџЋЇ Step 6: Quality Assurance & Final Video")
    st.write("**Comprehensive quality assessment and final video creation**")

    col1, col2 = st.columns(2)

    with col1:
        if st.button("рџЋЇ Generate Comprehensive QA Report", type="secondary"):
            if st.session_state.qa_framework:
                with st.spinner("рџЋЇ Generating comprehensive quality assessment..."):
                    parsed_chunks = st.session_state.parsed_chunk_data.get('parsed_chunks', [])
                    st.session_state.final_qa_report = st.session_state.qa_framework.generate_comprehensive_qa_report(
                        parsed_chunks, st.session_state.video_paths
                    )

                    st.success("вњ… Quality assurance report generated!")
```

```python
            else:
                st.error("QA Framework not initialized. Please complete narrative structure step.")

    with col2:
        if len(st.session_state.video_paths) == 8:
            if st.button("рџЋ¬ Create Final Feature Demo Video", type="primary"):
                with st.spinner("рџЋ¬ Creating final feature demonstration video..."):
                    result, error = create_feature_demonstration_final_video()

                    if result and not error:
                        st.success("рџЏ† Final feature demonstration video created!")
                        st.balloons()
                        st.video(result)

                        # Get file info
                        file_size = os.path.getsize(result) / (1024*1024)  # MB
                        st.success(f"**Final Video Created!**\n- Duration: 52 seconds (8 Г— 6.5s)\n- File size: {file_size:.1f} MB\n- Feature: {st.session_state.featur

                        with open(result, 'rb') as f:
                            feature_name = st.session_state.feature_analysis.get('feature_name', 'feature').replace(' ', '_').lower()
                            st.download_button(
                                "рџЋ¬ Download Feature Demo Video",
                                f.read(),
                                file_name=f"{feature_name}_demo_{datetime.now().strftime('%Y%m%d_%H%M%S')}.mp4",
                                mime="video/mp4"
                            )
                    else:
                        st.error(f"вќЊ Final video creation failed: {error}")
        else:
            st.warning(f"Need all 8 videos to create final video. Currently have: {len(st.session_state.video_paths)}/8")

# Display Comprehensive QA Report
if st.session_state.final_qa_report:
    st.subheader("рџЋЇ Comprehensive Quality Assurance Report")

    with st.expander("рџ“Љ Quality Assessment Results", expanded=True):
        report = st.session_state.final_qa_report

        # Overall score
        overall_score = report.get('overall_quality_score', 0)
        if overall_score >= 85:
            st.success(f"рџЏ† **Overall Quality Score: {overall_score}/100** - Excellent!")
        elif overall_score >= 70:
            st.warning(f"вљ пёЏ **Overall Quality Score: {overall_score}/100** - Good")
        else:
            st.error(f"вќЊ **Overall Quality Score: {overall_score}/100** - Needs improvement")

        # Detailed metrics
        col1, col2, col3 = st.columns(3)

        with col1:
            st.write("**рџ“– Narrative Analysis**")
            narrative = report.get('narrative_coherence', {})
            st.metric("Story Progression", f"{narrative.get('story_progression_score', 0)}/100")
            st.metric("Feature Integration", f"{narrative.get('feature_integration_score', 0)}/100")
            st.metric("Thematic Coherence", f"{narrative.get('thematic_coherence', 0)}/100")

        with col2:
            st.write("**рџ’« Emotional Engagement**")
            emotional = report.get('emotional_engagement', {})
            st.metric("Emotional Progression", f"{emotional.get('emotional_progression_score', 0)}/100")

            word_dist = emotional.get('emotional_word_distribution', {})
            st.write("**Emotional Word Analysis:**")
            st.write(f"вЂў Positive: {word_dist.get('positive', 0)} words")
            st.write(f"вЂў Negative: {word_dist.get('negative', 0)} words")
            st.write(f"вЂў Transition: {word_dist.get('transition', 0)} words")

            if emotional.get('emotional_peaks'):
                st.write("**Emotional Peaks:**")
                for peak in emotional.get('emotional_peaks', []):
                    st.write(f"вЂў {peak}")

        with col3:
            st.write("**рџЋҐ VEO3 Compatibility**")
            veo3 = report.get('veo3_compatibility', {})
            st.metric("Technical Compliance", f"{veo3.get('technical_compliance_score', 0)}/100")
            st.metric("Camera Quality", f"{veo3.get('camera_instruction_quality', 0)}/100")
            st.metric("Scene Diversity", f"{veo3.get('scene_diversity_score', 0)}/100")
            st.metric("Feature Demo Clarity", f"{veo3.get('feature_demonstration_clarity', 0)}/100")

        # Feature integration analysis
        st.write("**рџ”Ќ Feature Integration Analysis**")
        feature_integration = report.get('feature_integration_analysis', {})

        feature_col1, feature_col2 = st.columns(2)
        with feature_col1:
            st.info(f"**Feature Name:** {feature_integration.get('feature_name', 'N/A')}")
            st.metric("Value Propositions Integrated", feature_integration.get('value_propositions_integrated', 0))
            st.metric("Pain Points Addressed", feature_integration.get('pain_points_addressed', 0))

        with feature_col2:
            st.metric("Use Cases Incorporated", feature_integration.get('use_cases_incorporated', 0))

            # Video generation success
            video_success = report.get('video_generation_success', {})
            st.metric("Video Success Rate", f"{video_success.get('success_rate', 0):.1f}%")

            if video_success.get('failed_chunks'):
                st.warning(f"Failed chunks: {', '.join(map(str, video_success.get('failed_chunks', [])))}")

        # Improvement recommendations
        st.write("**рџЋЇ Improvement Recommendations**")
        recommendations = report.get('improvement_recommendations', [])
        if recommendations:
            for i, rec in enumerate(recommendations, 1):
                st.write(f"{i}. {rec}")
        else:
            st.success("рџЏ† No major improvements needed - Excellent quality!")

        # Download QA Report
        if st.button("рџ’ѕ Download QA Report"):
            timestamp = datetime.now().strftime('%Y%m%d_%H%M%S')
            feature_name = st.session_state.feature_analysis.get('feature_name', 'feature').replace(' ', '_').lower()

            report_json = json.dumps(report, indent=2)
            st.download_button(
                "рџ“Љ Download Complete QA Report (JSON)",
                report_json,
                file_name=f"qa_report_{feature_name}_{timestamp}.json",
                mime="application/json"
            )
```

```python
# Enhanced Progress Sidebar
st.sidebar.header("рџЋ¬ Feature Demo Pipeline Progress")

# Feature Analysis Progress
st.sidebar.subheader("рџ”Ќ Feature Analysis")
if st.session_state.feature_analysis:
    st.sidebar.success("вњ… Complete")
    feature_name = st.session_state.feature_analysis.get('feature_name', 'Unknown')
    st.sidebar.write(f"**Feature:** {feature_name[:30]}...")
    st.sidebar.write(f"**Value Props:** {len(st.session_state.feature_analysis.get('core_value_propositions', []))}")
    st.sidebar.write(f"**Pain Points:** {len(st.session_state.feature_analysis.get('customer_pain_points', []))}")
    st.sidebar.write(f"**Use Cases:** {len(st.session_state.feature_analysis.get('use_cases', []))}")
    st.sidebar.write(f"**Category:** {st.session_state.feature_analysis.get('feature_category', 'N/A')}")
else:
    st.sidebar.warning("вЏі Pending - Enter feature description")

# Feature Validation
if st.session_state.feature_validation:
    score = st.session_state.feature_validation.get('completeness_score', 0)
    if score >= 75:
        st.sidebar.success(f"вњ… Quality Score: {score}/100")
    elif score >= 50:
        st.sidebar.warning(f"вљ пёЏ Quality Score: {score}/100")
    else:
        st.sidebar.error(f"вќЊ Quality Score: {score}/100")

# Narrative Structure Progress
st.sidebar.subheader("рџ“– Narrative Structure")
if st.session_state.narrative_structure:
    st.sidebar.success("вњ… Complete")
    progression = st.session_state.narrative_structure.get('emotional_progression', {})
    st.sidebar.write(f"**Emotional Beats:** {len(progression)}/8")
    st.sidebar.write(f"**Use Case Integration:** вњ…")

    # Show emotional intensity range
    if progression:
        intensities = [beat.get('intensity', 5) for beat in progression.values()]
        st.sidebar.write(f"**Intensity Range:** {min(intensities)} - {max(intensities)}")
else:
    st.sidebar.warning("вЏі Pending - Complete feature analysis first")

# Story Generation Progress
st.sidebar.subheader("рџЋ¬ Story Generation")
if st.session_state.parsed_chunk_data.get('validation_passed'):
    st.sidebar.success("вњ… Complete")
    st.sidebar.write(f"**Chunks:** {st.session_state.parsed_chunk_data.get('chunk_count', 0)}/8")
    st.sidebar.write(f"**Character DNA:** {'вњ…' if st.session_state.parsed_chunk_data.get('character_dna') else 'вќЊ'}")
    st.sidebar.write(f"**Integration Score:** {st.session_state.parsed_chunk_data.get('feature_integration_score', 0)}/100")

    # Feature integration details
    validation_details = st.session_state.parsed_chunk_data.get('validation_details', {})
    st.sidebar.write(f"**Pain Points:** {validation_details.get('pain_points_count', 0)}")
    st.sidebar.write(f"**Capabilities:** {validation_details.get('capabilities_count', 0)}")
    st.sidebar.write(f"**Benefits:** {validation_details.get('benefits_count', 0)}")
else:
    st.sidebar.warning("вЏі Pending - Complete narrative structure first")

# Video Generation Progress
st.sidebar.subheader("рџЋҐ Video Generation")
st.sidebar.write(f"**Generated:** {len(st.session_state.video_paths)}/8")
st.sidebar.write(f"**Operations:** {len(st.session_state.operations)}/8")

if st.session_state.video_paths:
    total_duration = len(st.session_state.video_paths) * 6.5
    st.sidebar.write(f"**Total Duration:** {total_duration}s")
    st.sidebar.write(f"**Progress:** {(len(st.session_state.video_paths)/8)*100:.1f}%")

    # Show completion status
    completed_chunks = list(st.session_state.video_paths.keys())
    st.sidebar.write(f"**Completed:** {', '.join(map(str, sorted(completed_chunks)))}")

# QA Report Progress
st.sidebar.subheader("рџЋЇ Quality Assurance")
if st.session_state.final_qa_report:
    st.sidebar.success("вњ… Complete")
    score = st.session_state.final_qa_report.get('overall_quality_score', 0)
    if score >= 85:
        st.sidebar.success(f"рџЏ† Quality: {score}/100")
    elif score >= 70:
        st.sidebar.warning(f"вљ пёЏ Quality: {score}/100")
    else:
        st.sidebar.error(f"вќЊ Quality: {score}/100")

    # Show success rate
    video_success = st.session_state.final_qa_report.get('video_generation_success', {})
    success_rate = video_success.get('success_rate', 0)
    st.sidebar.metric("Success Rate", f"{success_rate:.1f}%")
else:
    st.sidebar.warning("вЏі Pending - Generate videos first")

# Documentation Files Summary
st.sidebar.subheader("рџ“Ѓ Generated Files")
if os.path.exists(ANALYSIS_OUTPUT_FOLDER):
    analysis_files = len([f for f in os.listdir(ANALYSIS_OUTPUT_FOLDER) if f.endswith('.json')])
    st.sidebar.write(f"**Analysis:** {analysis_files} files")

if os.path.exists(DOCUMENTATION_FOLDER):
    doc_files = len([f for f in os.listdir(DOCUMENTATION_FOLDER) if f.endswith(('.txt', '.json'))])
    st.sidebar.write(f"**Documentation:** {doc_files} files")

if os.path.exists(QA_REPORTS_FOLDER):
    qa_files = len([f for f in os.listdir(QA_REPORTS_FOLDER) if f.endswith('.json')])
    st.sidebar.write(f"**QA Reports:** {qa_files} files")

video_files = len([f for f in os.listdir(VIDEO_OUTPUT_FOLDER) if f.endswith('.mp4')]) if os.path.exists(VIDEO_OUTPUT_FOLDER) else 0
trimmed_files = len([f for f in os.listdir(TRIMMED_VIDEO_FOLDER) if f.endswith('.mp4')]) if os.path.exists(TRIMMED_VIDEO_FOLDER) else 0

st.sidebar.write(f"**Videos:** {video_files} original, {trimmed_files} trimmed")

# System Status
st.sidebar.subheader("вљ™пёЏ System Status")
st.sidebar.success("рџ”Ќ Feature Analysis Engine: Active")
st.sidebar.success("рџ“– Narrative Excellence: Active")
st.sidebar.success("рџ“‹ VEO3 Documentation: Active")
st.sidebar.success("рџЋЇ Quality Assurance: Active")
st.sidebar.success("вЏ±пёЏ Precise Timing Control: Active")
st.sidebar.success("рџЋ¬ Auto-Trim System: Active")

# API Status
st.sidebar.write("**API Connections:**")
if PERPLEXITY_API_KEY:
    st.sidebar.success("вњ… Perplexity AI")
if GEMINI_API_KEY:
    st.sidebar.success("вњ… Google Gemini")

# Footer
st.markdown("---")
st.markdown("**рџЋ¬ AI-Powered Cinematic Video Script Generator**")
st.markdown("*вњ… Feature Analysis Engine | вњ… Stakeholder Journey Mapping | вњ… Use Case Integration | вњ… Narrative Excellence | вњ… VEO3 Documentation | вњ… Qualit

# Performance metrics
if st.session_state.feature_analysis and st.session_state.video_paths:
    feature_name = st.session_state.feature_analysis.get('feature_name', 'Feature Demo')
    video_count = len(st.session_state.video_paths)
    total_duration = video_count * 6.5

    st.markdown(f"**рџ“Љ Session Summary:** {feature_name} | {video_count}/8 videos | {total_duration}s total | Generated: {datetime.now().strftime('%Y-%m-%d %H:%M')}'
```