AI-Orchestrated Video Production Pipeline

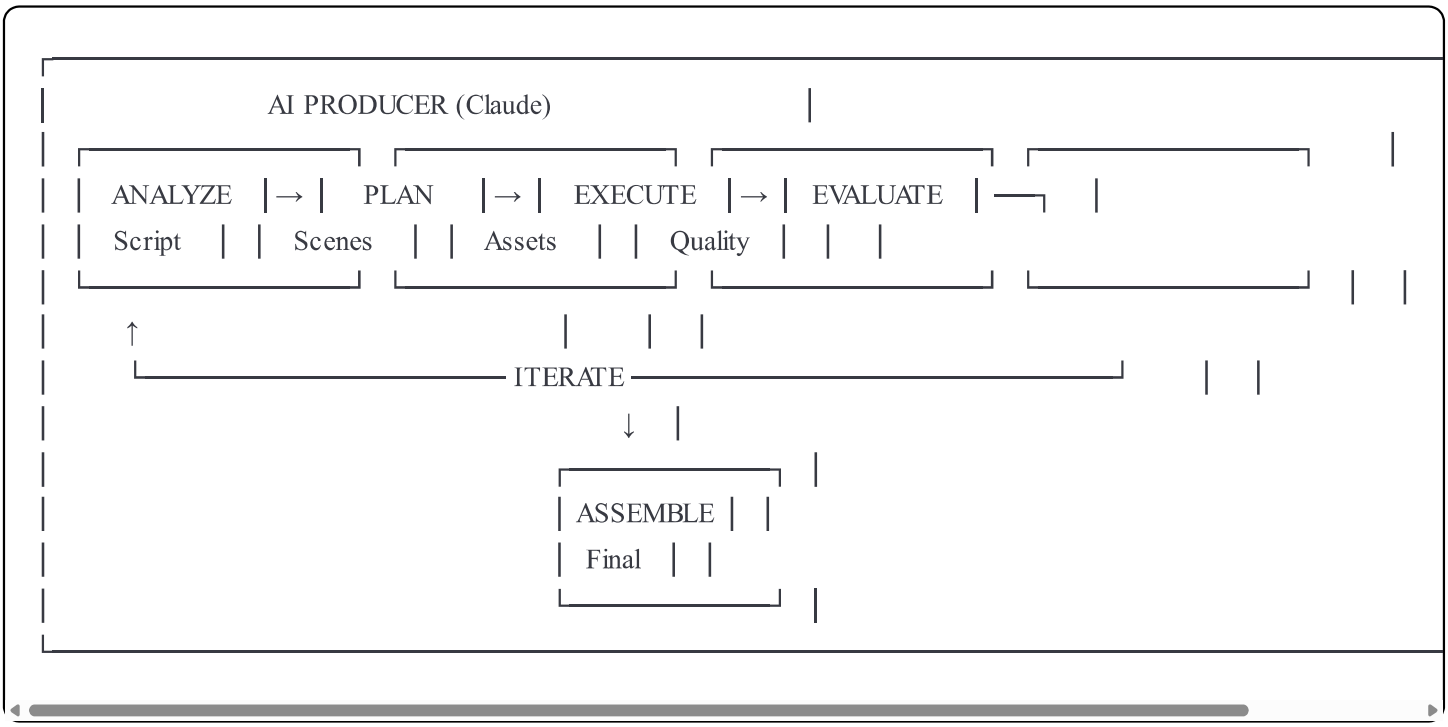
Technical Specification for Replit Agents

Executive Summary

This specification defines an **AI Producer** architecture where Claude (via Anthropic API) acts as the project lead for video creation. The AI Producer doesn't just execute tasks—it plans, evaluates, iterates, and makes creative decisions to ensure TV-quality output.

Core Architecture

The AI Producer Pattern



Phase 1: Script Analysis & Scene Breakdown

The AI Producer receives the script and creates a **Scene Manifest**—a structured document that breaks the video into discrete scenes with specific requirements.

```typescript
```

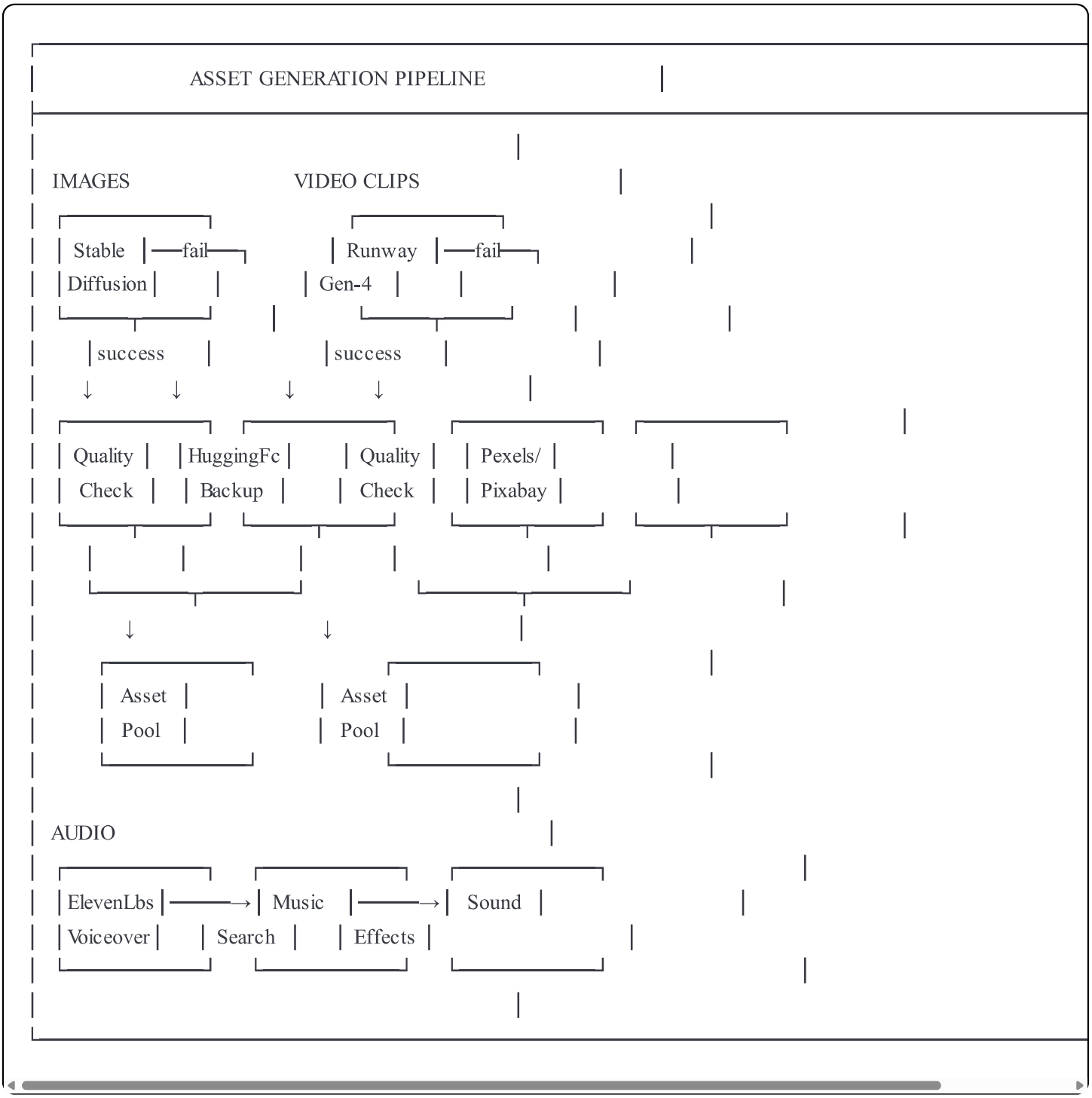
```typescript
interface SceneManifest {
  videoId: string;
  metadata: {
    title: string;
    duration: number;
    platform: 'youtube' | 'tiktok' | 'instagram' | 'linkedin';
    aspectRatio: '16:9' | '9:16' | '1:1';
    style: 'professional' | 'casual' | 'cinematic' | 'medical';
  };
  scenes: Scene[];
  globalAssets: {
    backgroundMusic: MusicRequirement[];
    colorGrading: ColorProfile;
    typography: TypographyProfile;
  };
}

interface Scene {
  id: string;
  section: 'HOOK' | 'PROBLEM' | 'SOLUTION' | 'SOCIAL_PROOF' | 'CTA';
  script: string;
  voiceover: {
    text: string;
    estimatedDuration: number;
    emotion: string;
    pacing: 'slow' | 'normal' | 'fast';
  };
  visualRequirements: {
    primary: VisualAsset;        // Main visual for this scene
    bRoll: VisualAsset[];        // Supporting footage
    overlays: OverlayAsset[];    // Text, graphics, etc.
    transitions: TransitionSpec;
  };
  audioRequirements: {
    soundEffects: SoundEffect[];
    musicIntensity: number;      // 0-100, relative to scene emotion
  };
  qualityCriteria: {
    mustHave: string[];          // Non-negotiable requirements
    niceToHave: string[];        // Enhancement opportunities
  };
}
```

```
interface VisualAsset {
  type: 'ai-generated' | 'stock-video' | 'stock-image' | 'animation' | 'product-shot';
  description: string;           // Detailed description for generation/search
  searchTerms: string[];         // For stock footage APIs
  aiPrompt?: string;             // For Runway/Stable Diffusion
  duration: number;
  priority: 'critical' | 'important' | 'optional';
  fallbackStrategy: string;      // What to do if primary fails
}
```

Phase 2: Asset Generation Pipeline

Each asset type has its own pipeline with built-in fallbacks:



Phase 3: Quality Evaluation

The AI Producer evaluates each generated asset against criteria:

typescript

```typescript
interface QualityEvaluation {
  assetId: string;
  scores: {
    relevance: number;          // 0-100: Does it match the scene requirement?
    technicalQuality: number;   // 0-100: Resolution, artifacts, etc.
    brandAlignment: number;     // 0-100: Matches video style/tone
    emotionalImpact: number;    // 0-100: Supports the narrative
  };
  overallScore: number;
  decision: 'accept' | 'regenerate' | 'use-fallback' | 'manual-review';
  reasoning: string;
  suggestedImprovements?: string[];
}
```

Phase 4: Assembly & Final Review

typescript

```typescript
interface AssemblyPlan {
  timeline: TimelineSegment[];
  renderSettings: {
    resolution: string;
    frameRate: number;
    codec: string;
    colorSpace: string;
  };
  finalChecks: {
    audioSync: boolean;
    captionSync: boolean;
    transitionSmooth: boolean;
    brandConsistent: boolean;
  };
}

interface TimelineSegment {
  startTime: number;
  endTime: number;
  layers: {
    video: AssetReference;
    audio: AssetReference[];
    captions: CaptionTrack;
    overlays: OverlayReference[];
  };
}
```

API Integration Specifications

1. Anthropic Claude API (AI Producer)

typescript

```
// The AI Producer uses structured prompts for each phase

const PRODUCER_SYSTEM_PROMPT = `
You are an AI Video Producer with expertise in creating professional marketing videos.
Your role is to:
1. Analyze scripts and create detailed scene breakdowns
2. Specify exact requirements for each visual and audio asset
3. Evaluate generated content against quality criteria
4. Make creative decisions to ensure cohesive, TV-quality output

You think like a human video producer but can process and evaluate content systematically.
Always provide structured JSON responses as specified in each request.
`;

async function analyzeScript(script: string, config: VideoConfig): Promise<SceneManifest> {
  const response = await anthropic.messages.create({
    model: 'claude-sonnet-4-20250514',
    max_tokens: 8000,
    system: PRODUCER_SYSTEM_PROMPT,
    messages: [{
      role: 'user',
      content: `
        Analyze this marketing script and create a detailed Scene Manifest.

        Script: ${script}

        Video Configuration:
        - Platform: ${config.platform}
        - Duration: ${config.duration}
        - Style: ${config.style}
        - Target Audience: ${config.audience}

        Return a JSON Scene Manifest with:
        1. Metadata for the video
        2. Detailed scene breakdown (one per script section)
        3. Specific visual requirements with AI prompts for generation
        4. Audio requirements including voiceover pacing and music intensity
        5. Quality criteria for each scene

        Be extremely specific in visual descriptions - these will be used
        directly as prompts for AI image/video generation.
      `
    }]
```

```
  });

  return JSON.parse(response.content[0].text);
}

async function evaluateAsset(
  asset: GeneratedAsset,
  requirement: VisualRequirement
): Promise<QualityEvaluation> {
  // For images, include base64 in the request
  const response = await anthropic.messages.create({
    model: 'claude-sonnet-4-20250514',
    max_tokens: 2000,
    system: PRODUCER_SYSTEM_PROMPT,
    messages: [{
      role: 'user',
      content: [
        {
          type: 'image',
          source: { type: 'base64', media_type: 'image/png', data: asset.base64 }
        },
        {
          type: 'text',
          text: `
            Evaluate this generated image against the requirement:

            Requirement: ${JSON.stringify(requirement)}

            Score on:
            - Relevance (0-100): Does it match what was requested?
            - Technical Quality (0-100): Resolution, artifacts, composition
            - Brand Alignment (0-100): Professional, matches intended style
            - Emotional Impact (0-100): Supports the marketing message

            Return JSON with scores, overall decision, and reasoning.
          `
        }
      ]
    }]
  });

  return JSON.parse(response.content[0].text);
}
```

2. Runway API (Video Generation)

```typescript
```

```typescript
interface RunwayConfig {
  apiKey: string;
  model: 'gen-4-turbo' | 'gen-3-alpha';
}

async function generateVideoClip(
  prompt: string,
  config: {
    duration: 4 | 10;  // Gen-4 supports 4s or 10s
    aspectRatio: '16:9' | '9:16' | '1:1';
    seed?: number;
    imagePrompt?: string;  // Optional image to animate
  }
): Promise<VideoAsset> {
  const runway = new RunwayClient(process.env.RUNWAY_API_KEY);

  // Gen-4 Turbo for fast clips, Gen-3 for higher quality
  const task = await runway.imageToVideo.create({
    model: 'gen-4-turbo',
    promptText: prompt,
    duration: config.duration,
    ratio: config.aspectRatio,
    ...(config.imagePrompt && { promptImage: config.imagePrompt })
  });

  // Poll for completion
  let result = await runway.tasks.retrieve(task.id);
  while (result.status === 'RUNNING' || result.status === 'PENDING') {
    await sleep(5000);
    result = await runway.tasks.retrieve(task.id);
  }

  if (result.status === 'SUCCEEDED') {
    return {
      url: result.output[0],
      duration: config.duration,
      metadata: { prompt, model: 'gen-4-turbo' }
    };
  }

  throw new Error(`Runway generation failed: ${result.failure}`);
}
```

3. Stable Diffusion / Stability AI (Image Generation)

typescript

```typescript
async function generateImage(
  prompt: string,
  config: {
    width: number;
    height: number;
    style?: 'photorealistic' | 'cinematic' | 'illustration';
    negativePrompt?: string;
  }
): Promise<ImageAsset> {
  const response = await fetch(
    'https://api.stability.ai/v2beta/stable-image/generate/sd3',
    {
      method: 'POST',
      headers: {
        'Authorization': `Bearer ${process.env.STABILITY_API_KEY}`,
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({
        prompt: prompt,
        negative_prompt: config.negativePrompt || 'blurry, low quality, distorted',
        aspect_ratio: getAspectRatio(config.width, config.height),
        output_format: 'png',
        style_preset: config.style || 'photorealistic'
      })
    }
  );

  const result = await response.json();
  return {
    base64: result.image,
    width: config.width,
    height: config.height,
    metadata: { prompt, style: config.style }
  };
}
```

4. ElevenLabs (Voiceover)

typescript

```typescript
async function generateVoiceover(
  text: string,
  config: {
    voiceId: string;
    stability?: number;       // 0-1, higher = more consistent
    similarityBoost?: number; // 0-1, higher = more similar to original voice
    style?: number;           // 0-1, style exaggeration
  }
): Promise<AudioAsset> {
  const response = await fetch(
    `https://api.elevenlabs.io/v1/text-to-speech/${config.voiceId}`,
    {
      method: 'POST',
      headers: {
        'xi-api-key': process.env.ELEVENLABS_API_KEY,
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({
        text: text,
        model_id: 'eleven_multilingual_v2',
        voice_settings: {
          stability: config.stability || 0.5,
          similarity_boost: config.similarityBoost || 0.75,
          style: config.style || 0.3
        }
      })
    }
  );

  const audioBuffer = await response.arrayBuffer();
  return {
    buffer: Buffer.from(audioBuffer),
    format: 'mp3',
    metadata: { text, voiceId: config.voiceId }
  };
}
```

5. Stock Footage (Pexels/Pixabay/Unsplash)

typescript

```
async function searchStockVideo(
  query: string,
  config: {
    minDuration?: number;
    maxDuration?: number;
    orientation?: 'landscape' | 'portrait' | 'square';
    providers: ('pexels' | 'pixabay')[];
  }
): Promise<StockAsset[]> {
  const results: StockAsset[] = [];

  // Search multiple providers in parallel
  const searches = config.providers.map(async (provider) => {
    if (provider === 'pexels') {
      const response = await fetch(
        `https://api.pexels.com/videos/search?query=${encodeURIComponent(query)}&per_page=10`,
        { headers: { 'Authorization': process.env.PEXELS_API_KEY } }
      );
      const data = await response.json();
      return data.videos.map(v => ({
        provider: 'pexels',
        id: v.id,
        url: v.video_files[0].link,
        duration: v.duration,
        preview: v.image,
        attribution: v.user.name
      }));
    }

    if (provider === 'pixabay') {
      const response = await fetch(
        `https://pixabay.com/api/videos/?key=${process.env.PIXABAY_API_KEY}&q=${encodeURIComponent(query)}&per_
      );
      const data = await response.json();
      return data.hits.map(v => ({
        provider: 'pixabay',
        id: v.id,
        url: v.videos.medium.url,
        duration: v.duration,
        preview: v.picture_id,
        attribution: v.user
      }));
    }
```

```
  });

  const allResults = await Promise.all(searches);
  return allResults.flat().filter(r =>
    (!config.minDuration || r.duration >= config.minDuration) &&
    (!config.maxDuration || r.duration <= config.maxDuration)
  );
}
```

6. Hugging Face (Fallback Generation)

typescript

```
async function generateImageHuggingFace(
  prompt: string,
  model: string = 'stabilityai/stable-diffusion-xl-base-1.0'
): Promise<ImageAsset> {
  const response = await fetch(
    `https://api-inference.huggingface.co/models/${model}`,
    {
      method: 'POST',
      headers: {
        'Authorization': `Bearer ${process.env.HUGGINGFACE_API_TOKEN}`,
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({ inputs: prompt })
    }
  );

  const buffer = await response.arrayBuffer();
  return {
    base64: Buffer.from(buffer).toString('base64'),
    metadata: { prompt, model }
  };
}
```

User Interface Enhancements

Workflow Visualization Component

tsx

```tsx
// WorkflowProgress.tsx
interface WorkflowStage {
  id: string;
  name: string;
  status: 'pending' | 'active' | 'complete' | 'error';
  progress?: number;
  details?: string;
  assets?: AssetPreview[];
}

const WorkflowProgress: React.FC<{ stages: WorkflowStage[] }> = ({ stages }) => {
  return (
    <div className="workflow-container">
      {stages.map((stage, index) => (
        <div key={stage.id} className={`workflow-stage ${stage.status}`}>
          <div className="stage-header">
            <div className="stage-number">{index + 1}</div>
            <div className="stage-name">{stage.name}</div>
            <StatusIndicator status={stage.status} />
          </div>

          {stage.status === 'active' && stage.progress !== undefined && (
            <ProgressBar value={stage.progress} />
          )}

          {stage.details && (
            <div className="stage-details">{stage.details}</div>
          )}

          {stage.assets && stage.assets.length > 0 && (
            <div className="asset-previews">
              {stage.assets.map(asset => (
                <AssetPreviewCard
                  key={asset.id}
                  asset={asset}
                  onRegenerate={() => handleRegenerate(asset)}
                  onApprove={() => handleApprove(asset)}
                />
              ))}
            </div>
          )}
        </div>
      ))}
```

```
    </div>
  );
};
```

Scene Editor Component

tsx

```tsx
// SceneEditor.tsx - Allow users to view and modify AI decisions
interface SceneEditorProps {
  scene: Scene;
  assets: GeneratedAsset[];
  onAssetChange: (sceneId: string, assetType: string, newAsset: AssetSpec) => void;
  onRegenerateRequest: (sceneId: string, assetType: string) => void;
}

const SceneEditor: React.FC<SceneEditorProps> = ({
  scene,
  assets,
  onAssetChange,
  onRegenerateRequest
}) => {
  return (
    <div className="scene-editor">
      <div className="scene-header">
        <span className="section-badge">{scene.section}</span>
        <span className="duration">{scene.voiceover.estimatedDuration}s</span>
      </div>

      <div className="script-preview">
        <p>{scene.script}</p>
      </div>

      <div className="visual-panel">
        <h4>Primary Visual</h4>
        <AssetSelector
          current={assets.find(a => a.type === 'primary')}
          requirement={scene.visualRequirements.primary}
          alternatives={assets.filter(a => a.type === 'primary-alternative')}
          onSelect={(asset) => onAssetChange(scene.id, 'primary', asset)}
          onRegenerate={() => onRegenerateRequest(scene.id, 'primary')}
        />

        <h4>B-Roll Options</h4>
        <div className="broll-grid">
          {scene.visualRequirements.bRoll.map((req, i) => (
            <AssetSelector
              key={i}
              current={assets.find(a => a.type === `broll-${i}`)}
              requirement={req}
              onSelect={(asset) => onAssetChange(scene.id, `broll-${i}`, asset)}
```

```
            onRegenerate={() => onRegenerateRequest(scene.id, `broll-${i}`)}
          />
        ))}
      </div>
    </div>

    <div className="audio-panel">
      <h4>Voiceover</h4>
      <AudioPreview
        audio={assets.find(a => a.type === 'voiceover')}
        text={scene.voiceover.text}
      />
      <VoiceSettings
        emotion={scene.voiceover.emotion}
        pacing={scene.voiceover.pacing}
      />
    </div>
  </div>
 );
};
```

Real-Time Production Log

tsx

```
// ProductionLog.tsx - Show what the AI Producer is doing
interface LogEntry {
  timestamp: Date;
  type: 'decision' | 'generation' | 'evaluation' | 'error' | 'success';
  message: string;
  details?: any;
}

const ProductionLog: React.FC<{ entries: LogEntry[] }> = ({ entries }) => {
  return (
    <div className="production-log">
      <h3>🎬 AI Producer Activity</h3>
      <div className="log-entries">
        {entries.map((entry, i) => (
          <div key={i} className={`log-entry ${entry.type}`}>
            <span className="timestamp">
              {entry.timestamp.toLocaleTimeString()}
            </span>
            <span className="icon">{getIcon(entry.type)}</span>
            <span className="message">{entry.message}</span>
            {entry.details && (
              <details>
                <summary>Details</summary>
                <pre>{JSON.stringify(entry.details, null, 2)}</pre>
              </details>
            )}
          </div>
        ))}
      </div>
    </div>
  );
};
```

Complete Production Flow

Main Production Controller

typescript

```typescript
// videoProducer.ts - Main orchestration class

export class AIVideoProducer {
  private anthropic: AnthropicClient;
  private assetGenerators: AssetGenerators;
  private eventEmitter: EventEmitter;

  constructor(config: ProducerConfig) {
    this.anthropic = new AnthropicClient(config.anthropicKey);
    this.assetGenerators = {
      runway: new RunwayGenerator(config.runwayKey),
      stability: new StabilityGenerator(config.stabilityKey),
      elevenlabs: new ElevenLabsGenerator(config.elevenLabsKey),
      stock: new StockSearcher({
        pexels: config.pexelsKey,
        pixabay: config.pixabayKey,
        unsplash: config.unsplashKey
      }),
      huggingface: new HuggingFaceGenerator(config.huggingfaceToken)
    };
    this.eventEmitter = new EventEmitter();
  }

  async produceVideo(
    script: string,
    config: VideoConfig
  ): Promise<ProductionResult> {
    const productionId = generateId();

    try {
      // Phase 1: Script Analysis
      this.emit('phase', { phase: 'analysis', status: 'started' });
      const manifest = await this.analyzeScript(script, config);
      this.emit('phase', { phase: 'analysis', status: 'complete', manifest });

      // Phase 2: Asset Generation
      this.emit('phase', { phase: 'generation', status: 'started' });
      const assets = await this.generateAllAssets(manifest);
      this.emit('phase', { phase: 'generation', status: 'complete', assets });

      // Phase 3: Quality Evaluation
      this.emit('phase', { phase: 'evaluation', status: 'started' });
      const evaluated = await this.evaluateAssets(assets, manifest);
```

```
    this.emit('phase', { phase: 'evaluation', status: 'complete', evaluated });

    // Phase 4: Handle Failed Assets
    if (evaluated.failures.length > 0) {
      this.emit('phase', { phase: 'regeneration', status: 'started' });
      const regenerated = await this.handleFailures(evaluated.failures, manifest);
      this.emit('phase', { phase: 'regeneration', status: 'complete', regenerated });
    }

    // Phase 5: Assembly
    this.emit('phase', { phase: 'assembly', status: 'started' });
    const assembled = await this.assembleVideo(evaluated.approved, manifest);
    this.emit('phase', { phase: 'assembly', status: 'complete', assembled });

    // Phase 6: Final Review
    this.emit('phase', { phase: 'review', status: 'started' });
    const finalReview = await this.finalReview(assembled, manifest);
    this.emit('phase', { phase: 'review', status: 'complete', finalReview });

    return {
      success: true,
      videoUrl: assembled.outputUrl,
      manifest,
      assets: evaluated.approved,
      quality: finalReview
    };

  } catch (error) {
    this.emit('error', { productionId, error });
    throw error;
  }
}

private async analyzeScript(
  script: string,
  config: VideoConfig
): Promise<SceneManifest> {
  this.emit('decision', {
    type: 'analyzing_script',
    message: 'AI Producer is analyzing the script and planning scenes...'
  });

  const response = await this.anthropic.messages.create({
    model: 'claude-sonnet-4-20250514',
```

```
    max_tokens: 8000,
    system: AI_PRODUCER_SYSTEM_PROMPT,
    messages: [{
      role: 'user',
      content: `
        Create a detailed Scene Manifest for this marketing video.

        SCRIPT:
        ${script}

        CONFIGURATION:
        - Platform: ${config.platform} (${PLATFORM_SPECS[config.platform]})
        - Target Duration: ${config.duration} seconds
        - Style: ${config.style}
        - Audience: ${config.targetAudience || 'General'}

        For each scene, specify:
        1. Exact visual requirements with detailed AI generation prompts
        2. B-roll suggestions with search terms for stock footage
        3. Voiceover settings (emotion, pacing, emphasis)
        4. Transition types between scenes
        5. Quality criteria that the generated assets must meet

        Think like a professional video producer. Make creative decisions
        that will result in a cohesive, engaging, TV-quality video.

        Return ONLY valid JSON matching the SceneManifest schema.
      `
    }]
  });

  const manifest = JSON.parse(response.content[0].text);

  this.emit('decision', {
    type: 'script_analyzed',
    message: `Created ${manifest.scenes.length} scenes with ${this.countAssets(manifest)} total asset requirements`,
    details: manifest
  });

  return manifest;
}

private async generateAllAssets(manifest: SceneManifest): Promise<GeneratedAssets> {
  const assets: GeneratedAssets = {
```

```
  images: [],
  videos: [],
  audio: [],
  stock: []
};

// Generate voiceover first (needed for timing)
for (const scene of manifest.scenes) {
  this.emit('generation', {
    type: 'voiceover',
    scene: scene.id,
    message: `Generating voiceover for ${scene.section}...`
  });

  const voiceover = await this.assetGenerators.elevenlabs.generate(
    scene.voiceover.text,
    {
      emotion: scene.voiceover.emotion,
      pacing: scene.voiceover.pacing
    }
  );

  assets.audio.push({
    sceneId: scene.id,
    type: 'voiceover',
    ...voiceover
  });
}

// Generate visuals in parallel
const visualPromises = manifest.scenes.flatMap(scene => [
  // Primary visual
  this.generatePrimaryVisual(scene).then(v => ({ sceneId: scene.id, type: 'primary', ...v })),
  // B-roll
  ...scene.visualRequirements.bRoll.map((req, i) =>
    this.generateBRoll(req).then(v => ({ sceneId: scene.id, type: `broll-${i}`, ...v }))
  )
]);

const visuals = await Promise.allSettled(visualPromises);

for (const result of visuals) {
  if (result.status === 'fulfilled') {
    if (result.value.isVideo) {
```

```
          assets.videos.push(result.value);
        } else {
          assets.images.push(result.value);
        }
      } else {
        this.emit('generation_error', {
          error: result.reason,
          message: 'Asset generation failed, will use fallback'
        });
      }
    }

    return assets;
  }

  private async generatePrimaryVisual(scene: Scene): Promise<VisualAsset> {
    const req = scene.visualRequirements.primary;

    // Try primary generator based on type
    try {
      if (req.type === 'ai-generated' && req.aiPrompt) {
        // Try Runway for video, Stability for image
        if (scene.voiceover.estimatedDuration > 3) {
          return await this.assetGenerators.runway.generate(req.aiPrompt, {
            duration: Math.min(scene.voiceover.estimatedDuration, 10),
            aspectRatio: this.manifest.metadata.aspectRatio
          });
        } else {
          return await this.assetGenerators.stability.generate(req.aiPrompt, {
            width: 1920,
            height: 1080
          });
        }
      }

      if (req.type === 'stock-video' || req.type === 'stock-image') {
        const results = await this.assetGenerators.stock.search(req.searchTerms.join(' '), {
          type: req.type.includes('video') ? 'video' : 'image'
        });
        return results[0]; // Let evaluation pick the best
      }
    } catch (error) {
      // Fallback strategy
      this.emit('decision', {
```

```
      type: 'fallback',
      message: `Primary generation failed for ${scene.id}, using fallback: ${req.fallbackStrategy}`
    });

    return await this.executeFallback(req);
  }
}

private async evaluateAssets(
  assets: GeneratedAssets,
  manifest: SceneManifest
): Promise<EvaluationResult> {
  const approved: EvaluatedAsset[] = [];
  const failures: FailedAsset[] = [];

  // Evaluate each asset with Claude
  for (const asset of [...assets.images, ...assets.videos]) {
    const scene = manifest.scenes.find(s => s.id === asset.sceneId);
    const requirement = this.getRequirement(scene, asset.type);

    this.emit('evaluation', {
      asset: asset.id,
      message: `Evaluating ${asset.type} for ${scene.section}...`
    });

    const evaluation = await this.anthropic.messages.create({
      model: 'claude-sonnet-4-20250514',
      max_tokens: 1000,
      messages: [{
        role: 'user',
        content: [
          asset.base64 ? {
            type: 'image',
            source: { type: 'base64', media_type: 'image/png', data: asset.base64 }
          } : {
            type: 'text',
            text: `Video asset URL: ${asset.url}`
          },
          {
            type: 'text',
            text: `
              Evaluate this asset for use in a ${scene.section} scene.

              REQUIREMENT: ${JSON.stringify(requirement)}
```

```
            SCENE CONTEXT: ${scene.script}

            Score on:
            - Relevance (0-100)
            - Technical Quality (0-100)
            - Brand Alignment (0-100)
            - Emotional Impact (0-100)

            Assets need 70+ overall to be approved.

            Return JSON: { scores: {...}, overall: number, decision: string, reasoning: string }
          `
        }
      ]
    }]
  });

  const result = JSON.parse(evaluation.content[0].text);

  if (result.decision === 'accept') {
    approved.push({ ...asset, evaluation: result });
  } else {
    failures.push({ ...asset, evaluation: result });
  }

  this.emit('evaluation_result', {
    asset: asset.id,
    score: result.overall,
    decision: result.decision,
    reasoning: result.reasoning
  });
 }

 return { approved, failures };
}

private async assembleVideo(
 assets: EvaluatedAsset[],
 manifest: SceneManifest
): Promise<AssembledVideo> {
 // Create timeline
 const timeline = this.buildTimeline(assets, manifest);

 // Use FFmpeg or similar for assembly
```

```
    // This would integrate with a video processing service

    this.emit('assembly', {
      message: 'Building final video timeline...',
      segments: timeline.length
    });

    // ... assembly implementation

    return {
      outputUrl: `https://storage.example.com/videos/${manifest.videoId}.mp4`,
      duration: timeline.reduce((sum, seg) => sum + seg.duration, 0),
      timeline
    };
  }

  // Event subscription for UI updates
  on(event: string, callback: Function) {
    this.eventEmitter.on(event, callback);
  }

  private emit(event: string, data: any) {
    this.eventEmitter.emit(event, { timestamp: new Date(), ...data });
  }
}
```

Database Schema for Production State

sql

```sql
-- Store production jobs and their state
CREATE TABLE video_productions (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  user_id UUID REFERENCES users(id),
  status VARCHAR(50) NOT NULL DEFAULT 'pending',
  script TEXT NOT NULL,
  config JSONB NOT NULL,
  manifest JSONB,
  created_at TIMESTAMP DEFAULT NOW(),
  updated_at TIMESTAMP DEFAULT NOW(),
  completed_at TIMESTAMP
);

-- Store individual assets for each production
CREATE TABLE production_assets (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  production_id UUID REFERENCES video_productions(id),
  scene_id VARCHAR(100) NOT NULL,
  asset_type VARCHAR(50) NOT NULL,
  source VARCHAR(50) NOT NULL, -- 'runway', 'stability', 'pexels', etc.
  url TEXT,
  base64_preview TEXT,
  metadata JSONB,
  evaluation JSONB,
  status VARCHAR(50) DEFAULT 'pending',
  created_at TIMESTAMP DEFAULT NOW()
);

-- Store production events for logging/debugging
CREATE TABLE production_events (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  production_id UUID REFERENCES video_productions(id),
  event_type VARCHAR(50) NOT NULL,
  message TEXT,
  details JSONB,
  created_at TIMESTAMP DEFAULT NOW()
);
```

Key Implementation Notes for Replit Agents

1. **Always Start with Claude Analysis**: Never generate assets without first having Claude create a detailed manifest. This ensures coherent creative direction.

2. **Parallel Generation with Fallbacks**: Generate multiple assets in parallel but have fallback chains (Stability → HuggingFace → Stock).
3. **Visual Quality Gates**: Always run generated images/videos through Claude evaluation before assembly. Reject and regenerate if below threshold.
4. **Real-Time UI Updates**: Use WebSocket or SSE to stream production events to the frontend. Users should see exactly what the AI Producer is doing.
5. **User Intervention Points**: After each major phase, allow users to review and modify AI decisions before proceeding.
6. **Asset Caching**: Store generated assets in a CDN/S3 for reuse and quick regeneration.
7. **Error Recovery**: If any phase fails, save state and allow resumption from the last checkpoint.

Recommended File Structure

```
/video-producer
  /api
    /routes
      production.ts      # REST endpoints for video production
    /services
      producer.ts        # AIVideoProducer class
      generators/
        runway.ts
        stability.ts
        elevenlabs.ts
        stock.ts
        huggingface.ts
      evaluator.ts       # Claude-based quality evaluation
      assembler.ts       # Video assembly with FFmpeg
  /components
    WorkflowProgress.tsx
    SceneEditor.tsx
    AssetSelector.tsx
    ProductionLog.tsx
    VideoPreview.tsx
  /hooks
    useProduction.ts     # WebSocket connection for real-time updates
  /types
    manifest.ts          # All TypeScript interfaces
  /prompts
```

```
producer.ts          # Claude system prompts
evaluator.ts         # Evaluation prompts
```

This architecture ensures Claude acts as a true "producer" making creative decisions, not just executing commands.