Replit Agent Implementation Instructions

AI-Orchestrated Video Production Pipeline

CRITICAL CONTEXT

You are implementing a video production system where **Claude (Anthropic API) acts as the Project Lead**. Claude doesn't just execute—it **plans, evaluates, and makes creative decisions**.

The current implementation has individual features that don't coordinate. Your job is to build the orchestration layer.

IMPLEMENTATION PRIORITY ORDER

Phase 1: Scene Manifest System (Do This First)

The entire system hinges on Claude creating a structured **Scene Manifest** from the script. Without this, everything else is disconnected.

```typescript
```

```
// /api/services/manifestGenerator.ts

import Anthropic from '@anthropic-ai/sdk';

const MANIFEST_SYSTEM_PROMPT = `You are an AI Video Producer. When given a marketing script, you create a detai

Your manifest must be executable—every visual requirement must include:
1. A detailed AI generation prompt (for Runway/Stability)
2. Search terms for stock footage fallback
3. Quality criteria to evaluate the result

Think like a professional video producer making creative decisions.`;

export async function generateManifest(
  script: string,
  config: {
    platform: 'youtube' | 'tiktok' | 'instagram' | 'linkedin';
    duration: number;
    style: 'professional' | 'casual' | 'cinematic' | 'medical';
    targetAudience?: string;
  }
): Promise<SceneManifest> {
  const client = new Anthropic({ apiKey: process.env.ANTHROPIC_API_KEY });

  const response = await client.messages.create({
    model: 'claude-sonnet-4-20250514',
    max_tokens: 8000,
    system: MANIFEST_SYSTEM_PROMPT,
    messages: [{
      role: 'user',
      content: `Create a Scene Manifest for this video:

SCRIPT:
${script}

CONFIGURATION:
- Platform: ${config.platform}
- Duration: ${config.duration} seconds
- Style: ${config.style}
- Audience: ${config.targetAudience || 'General'}

Return ONLY valid JSON matching this structure:
{
  "videoId": "unique-id",
  "metadata": { "title": "", "duration": 0, "platform": "", "aspectRatio": "", "style": "" },
  "scenes": [
    {
      "id": "scene-1",
      "section": "HOOK",
      "script": "The exact script text for this section",
      "voiceover": {
        "text": "Voiceover text",
        "estimatedDuration": 5,
        "emotion": "intrigued",
        "pacing": "normal"
      },
      "visualRequirements": {
        "primary": {
          "type": "ai-generated",
          "description": "Detailed description",
          "aiPrompt": "Exact prompt for Runway/Stability",
          "searchTerms": ["fallback", "search", "terms"],
          "duration": 5,
          "priority": "critical",
          "fallbackStrategy": "search stock footage for..."
        },
        "bRoll": [],
        "overlays": [],
        "transitions": { "in": "fade", "out": "crossfade" }
      },
      "qualityCriteria": {
        "mustHave": ["professional appearance", "matches brand"],
        "niceToHave": ["motion", "engaging composition"]
      }
    }
  ],
  "globalAssets": {
    "backgroundMusic": { "mood": "uplifting", "intensity": "moderate" },
    "colorGrading": "natural",
    "typography": { "font": "clean sans-serif", "style": "minimal" }
  }
}`
    }]
  });
  return JSON.parse(response.content[0].text);
}
```

Phase 2: Asset Generation with Fallbacks

Each asset type needs a primary generator and fallback chain:

```
typescript
```

```
// /api/services/assetGenerator.ts

export class AssetGenerator {
  private runway: RunwayClient;
  private stability: StabilityClient;
  private huggingface: HuggingFaceClient;
  private pexels: PexelsClient;
  private elevenlabs: ElevenLabsClient;

  constructor() {
    this.runway = new RunwayClient(process.env.RUNWAY_API_KEY);
    this.stability = new StabilityClient(process.env.STABILITY_API_KEY);
    this.huggingface = new HuggingFaceClient(process.env.HUGGINGFACE_API_TOKEN);
    this.pexels = new PexelsClient(process.env.PEXELS_API_KEY);
    this.elevenlabs = new ElevenLabsClient(process.env.ELEVENLABS_API_KEY);
  }

  async generateVisual(requirement: VisualRequirement): Promise<GeneratedAsset> {
    const errors: Error[] = [];

    // Try primary: AI generation
    if (requirement.type === 'ai-generated' && requirement.aiPrompt) {
      try {
        // For longer durations, use Runway video
        if (requirement.duration > 3) {
          return await this.generateRunwayVideo(requirement);
        }
        // For stills, use Stability
        return await this.generateStabilityImage(requirement);
      } catch (e) {
        errors.push(e);
        console.log(`Primary generation failed: ${e.message}`);
      }
    }

    // Fallback 1: HuggingFace
    try {
      return await this.generateHuggingFaceImage(requirement);
    } catch (e) {
      errors.push(e);
      console.log(`HuggingFace fallback failed: ${e.message}`);
    }
    // Fallback 2: Stock footage
    try {
      return await this.searchStock(requirement);
    } catch (e) {
      errors.push(e);
      console.log(`Stock fallback failed: ${e.message}`);
    }

    throw new Error(`All generation methods failed: ${errors.map(e => e.message).join(', ')}`);
  }

  private async generateRunwayVideo(req: VisualRequirement): Promise<GeneratedAsset> {
    // Runway Gen-4 Turbo API
    const task = await fetch('https://api.dev.runwayml.com/v1/image_to_video', {
      method: 'POST',
      headers: {
        'Authorization': `Bearer ${process.env.RUNWAY_API_KEY}`,
        'Content-Type': 'application/json',
        'X-Runway-Version': '2024-11-06'
      },
      body: JSON.stringify({
        model: 'gen4_turbo',
        promptText: req.aiPrompt,
        duration: Math.min(req.duration, 10),
        ratio: '16:9' // or '9:16' for vertical
      })
    });

    const { id } = await response.json();

    // Poll for completion
    let result;
    do {
      await sleep(5000);
      const pollRes = await fetch(`https://api.dev.runwayml.com/v1/tasks/${id}`, {
        headers: { 'Authorization': `Bearer ${process.env.RUNWAY_API_KEY}` }
      });
      result = await pollRes.json();
    } while (result.status === 'RUNNING' || result.status === 'PENDING');

    if (result.status !== 'SUCCEEDED') {
      throw new Error(`Runway failed: ${result.failure}`);
    }
    return {
      id: id,
      type: 'video',
      url: result.output[0],
      source: 'runway',
      metadata: { prompt: req.aiPrompt, duration: req.duration }
    };
  }

  private async generateStabilityImage(req: VisualRequirement): Promise<GeneratedAsset> {
    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',
        'Accept': 'application/json'
      },
      body: JSON.stringify({
        prompt: req.aiPrompt,
        negative_prompt: 'blurry, low quality, distorted, amateur',
        aspect_ratio: '16:9',
        output_format: 'png'
      })
    });

    const data = await response.json();

    return {
      id: crypto.randomUUID(),
      type: 'image',
      base64: data.image,
      source: 'stability',
      metadata: { prompt: req.aiPrompt }
    };
  }

  async generateVoiceover(text: string, config: VoiceConfig): Promise<AudioAsset> {
    const response = await fetch(`https://api.elevenlabs.io/v1/text-to-speech/${config.voiceId || 'pNInz6obpgDQGcFmaJgB'}
      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: 0.5,
          similarity_boost: 0.75,
          style: config.emotion === 'excited' ? 0.5 : 0.2
        }
      })
    });

    const buffer = await response.arrayBuffer();

    return {
      id: crypto.randomUUID(),
      type: 'audio',
      buffer: Buffer.from(buffer),
      source: 'elevenlabs',
      metadata: { text }
    };
  }
}
```

Phase 3: Quality Evaluation Gate

Every generated asset must pass Claude's evaluation before use:

```
typescript
```

```
// /api/services/qualityEvaluator.ts

export async function evaluateAsset(
  asset: GeneratedAsset,
  requirement: VisualRequirement,
  sceneContext: string
): Promise<QualityEvaluation> {
  const client = new Anthropic({ apiKey: process.env.ANTHROPIC_API_KEY });

  const content: MessageContent[] = [];

  // Include the image if we have it
  if (asset.base64) {
    content.push({
      type: 'image',
      source: { type: 'base64', media_type: 'image/png', data: asset.base64 }
    });
  }

  content.push({
    type: 'text',
    text: `Evaluate this generated asset for a marketing video.

REQUIREMENT:
${JSON.stringify(requirement, null, 2)}

SCENE CONTEXT:
${sceneContext}

Score each dimension 0-100:
- Relevance: Does it match what was requested?
- Technical Quality: Resolution, composition, no artifacts?
- Brand Alignment: Professional, matches intended style?
- Emotional Impact: Does it support the marketing message?

Assets need 70+ overall score to be approved.

Return JSON:
{
  "scores": { "relevance": 0, "technicalQuality": 0, "brandAlignment": 0, "emotionalImpact": 0 },
  "overall": 0,
  "decision": "accept" | "regenerate" | "use-fallback",
  "reasoning": "Explain your decision",
  "improvements": ["If regenerating, specific improvements needed"]
}`
  });

  const response = await client.messages.create({
    model: 'claude-sonnet-4-20250514',
    max_tokens: 1000,
    messages: [{ role: 'user', content }]
  });

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

Phase 4: Production Orchestrator

The main class that ties everything together:

```
typescript
```

```
// /api/services/videoProducer.ts

import { EventEmitter } from 'events';

export class VideoProducer extends EventEmitter {
  private manifestGenerator: ManifestGenerator;
  private assetGenerator: AssetGenerator;
  private qualityEvaluator: QualityEvaluator;

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

    // PHASE 1: Analyze Script
    this.emit('phase', { phase: 'analysis', status: 'started' });
    this.emit('log', { type: 'decision', message: 'AI Producer analyzing script...' });

    const manifest = await this.manifestGenerator.generate(script, config);

    this.emit('log', {
      type: 'decision',
      message: `Created ${manifest.scenes.length} scenes with detailed asset requirements`
    });
    this.emit('phase', { phase: 'analysis', status: 'complete', manifest });

    // PHASE 2: Generate All Assets
    this.emit('phase', { phase: 'generation', status: 'started' });

    const generatedAssets: GeneratedAsset[] = [];

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

      const voiceover = await this.assetGenerator.generateVoiceover(
        scene.voiceover.text,
        { emotion: scene.voiceover.emotion }
      );

      generatedAssets.push({ ...voiceover, sceneId: scene.id, assetType: 'voiceover' });
    }
```

```
// Generate visuals in parallel
const visualPromises = manifest.scenes.flatMap(scene => {
  const promises = [];

  // Primary visual
  promises.push(
    this.assetGenerator.generateVisual(scene.visualRequirements.primary)
      .then(asset => ({ ...asset, sceneId: scene.id, assetType: 'primary' }))
  );

  // B-roll
  scene.visualRequirements.bRoll.forEach((req, i) => {
    promises.push(
      this.assetGenerator.generateVisual(req)
        .then(asset => ({ ...asset, sceneId: scene.id, assetType: `broll-${i}` }))
    );
  });

  return promises;
});

const visualResults = await Promise.allSettled(visualPromises);

for (const result of visualResults) {
  if (result.status === 'fulfilled') {
    generatedAssets.push(result.value);
    this.emit('log', {
      type: 'generation',
      message: `Generated ${result.value.assetType} for scene ${result.value.sceneId}`
    });
  } else {
    this.emit('log', {
      type: 'error',
      message: `Generation failed: ${result.reason.message}`
    });
  }
}

this.emit('phase', { phase: 'generation', status: 'complete' });

// PHASE 3: Quality Evaluation
this.emit('phase', { phase: 'evaluation', status: 'started' });
```

```
    const approved: GeneratedAsset[] = [];
    const needsRegeneration: { asset: GeneratedAsset; evaluation: QualityEvaluation }[] = [];

    for (const asset of generatedAssets.filter(a => a.type !== 'audio')) {
      const scene = manifest.scenes.find(s => s.id === asset.sceneId);
      const requirement = this.getRequirement(scene, asset.assetType);

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

      const evaluation = await this.qualityEvaluator.evaluate(
        asset,
        requirement,
        scene.script
      );

      if (evaluation.decision === 'accept') {
        approved.push({ ...asset, evaluation });
        this.emit('log', {
          type: 'success',
          message: `✓ ${scene.section} ${asset.assetType}: ${evaluation.overall}/100 - Approved`
        });
      } else {
        needsRegeneration.push({ asset, evaluation });
        this.emit('log', {
          type: 'evaluation',
          message: `⚠ ${scene.section} ${asset.assetType}: ${evaluation.overall}/100 - ${evaluation.reasoning}`
        });
      }
    }

    // Add all audio assets (not evaluated visually)
    approved.push(...generatedAssets.filter(a => a.type === 'audio'));

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

    // PHASE 4: Handle Failures (Regenerate or Fallback)
    if (needsRegeneration.length > 0) {
      this.emit('phase', { phase: 'regeneration', status: 'started' });

      for (const { asset, evaluation } of needsRegeneration) {
        const scene = manifest.scenes.find(s => s.id === asset.sceneId);
      const requirement = this.getRequirement(scene, asset.assetType);

      this.emit('log', {
        type: 'generation',
        message: `Regenerating ${asset.assetType} with improvements: ${evaluation.improvements?.join(', ')}`
      });

      // Modify the prompt based on evaluation feedback
      const improvedRequirement = {
        ...requirement,
        aiPrompt: `${requirement.aiPrompt}. IMPORTANT: ${evaluation.improvements?.join('. ')}`
      };

      try {
        const regenerated = await this.assetGenerator.generateVisual(improvedRequirement);
        const reEvaluation = await this.qualityEvaluator.evaluate(regenerated, requirement, scene.script);

        if (reEvaluation.decision === 'accept') {
          approved.push({ ...regenerated, sceneId: asset.sceneId, assetType: asset.assetType, evaluation: reEvaluation });
          this.emit('log', { type: 'success', message: `✓ Regenerated asset approved: ${reEvaluation.overall}/100` });
        } else {
          // Use fallback
          this.emit('log', { type: 'generation', message: `Using stock fallback for ${asset.assetType}...` });
          const fallback = await this.assetGenerator.searchStock(requirement);
          approved.push({ ...fallback, sceneId: asset.sceneId, assetType: asset.assetType });
        }
      } catch (e) {
        this.emit('log', { type: 'error', message: `Regeneration failed, using placeholder` });
      }
    }

    this.emit('phase', { phase: 'regeneration', status: 'complete' });
  }

  // PHASE 5: Assembly
  this.emit('phase', { phase: 'assembly', status: 'started' });
  this.emit('log', { type: 'success', message: 'All assets approved. Assembling final video...' });

  const assembled = await this.assembleVideo(approved, manifest);

  this.emit('phase', { phase: 'assembly', status: 'complete', video: assembled });
  this.emit('log', { type: 'success', message: `Video complete! Duration: ${assembled.duration}s` });

  return {
      success: true,
      videoUrl: assembled.url,
      manifest,
      assets: approved
    };
  }
}
```

Phase 5: Real-Time UI Updates

Use Server-Sent Events or WebSocket to stream production progress:

```
typescript
```

```
// /api/routes/production.ts

import { Router } from 'express';
import { VideoProducer } from '../services/videoProducer';

const router = Router();

router.post('/produce', async (req, res) => {
  const { script, config } = req.body;

  // Set up SSE
  res.setHeader('Content-Type', 'text/event-stream');
  res.setHeader('Cache-Control', 'no-cache');
  res.setHeader('Connection', 'keep-alive');

  const producer = new VideoProducer();

  // Stream all events to client
  producer.on('phase', (data) => {
    res.write(`event: phase\ndata: ${JSON.stringify(data)}\n\n`);
  });

  producer.on('log', (data) => {
    res.write(`event: log\ndata: ${JSON.stringify({ ...data, timestamp: new Date() })}\n\n`);
  });

  try {
    const result = await producer.produceVideo(script, config);
    res.write(`event: complete\ndata: ${JSON.stringify(result)}\n\n`);
  } catch (error) {
    res.write(`event: error\ndata: ${JSON.stringify({ message: error.message })}\n\n`);
  }

  res.end();
});

export default router;
```

Phase 6: Frontend Integration

```
tsx
```

```tsx
// /components/VideoProducer.tsx

export function VideoProducer({ script, config }: Props) {
  const [phases, setPhases] = useState<Phase[]>([]);
  const [logs, setLogs] = useState<LogEntry[]>([]);
  const [result, setResult] = useState<ProductionResult | null>(null);

  const startProduction = useCallback(() => {
    const eventSource = new EventSource(`/api/produce?script=${encodeURIComponent(script)}&config=${encodeURICo

    eventSource.addEventListener('phase', (e) => {
      const data = JSON.parse(e.data);
      setPhases(prev => {
        const existing = prev.findIndex(p => p.phase === data.phase);
        if (existing >= 0) {
          const updated = [...prev];
          updated[existing] = data;
          return updated;
        }
        return [...prev, data];
      });
    });

    eventSource.addEventListener('log', (e) => {
      const data = JSON.parse(e.data);
      setLogs(prev => [...prev, data]);
    });

    eventSource.addEventListener('complete', (e) => {
      const data = JSON.parse(e.data);
      setResult(data);
      eventSource.close();
    });

    eventSource.addEventListener('error', () => {
      eventSource.close();
    });
  }, [script, config]);

  return (
    <div className="producer-container">
      <WorkflowProgress phases={phases} />
      <ProductionLog logs={logs} />
      {result && <VideoPreview video={result} />}
      <button onClick={startProduction}>Start Production</button>
    </div>
  );
}
```

KEY IMPLEMENTATION RULES

1. Claude Is Always The Decision Maker

- Never generate assets without Claude first creating a manifest
- Never use assets without Claude evaluating them
- Let Claude modify prompts based on evaluation feedback

2. Always Have Fallbacks

```
Primary: Runway/Stability AI
  ↓ failure
Fallback 1: HuggingFace
  ↓ failure
Fallback 2: Stock footage (Pexels/Pixabay)
  ↓ failure
Fallback 3: Static placeholder with text
```

3. Stream Everything to UI

- Users must see every decision the AI makes
- Show asset previews as they're generated
- Allow users to override AI decisions

4. Quality Gates Are Non-Negotiable

- Every visual asset needs 70+ score
- Failed assets trigger regeneration with improvements
- Track why assets fail to improve prompts

5. Scene Manifest Is Sacred

- The manifest defines the entire video structure
- All assets reference their scene ID
- Assembly follows the manifest timeline exactly

ENVIRONMENT VARIABLES REQUIRED

```env
ANTHROPIC_API_KEY=        # Claude for orchestration
RUNWAY_API_KEY=           # Video generation
STABILITY_API_KEY=        # Image generation
ELEVENLABS_API_KEY=       # Voiceover
HUGGINGFACE_API_TOKEN=    # Fallback image generation
PEXELS_API_KEY=           # Stock video
PIXABAY_API_KEY=          # Stock video/images
UNSPLASH_ACCESS_KEY=      # Stock images
```

TESTING CHECKLIST

- [ ] Manifest generates correctly from Pine Hill Farm script
- [ ] All 5 scenes have visual requirements with AI prompts
- [ ] Voiceovers generate with correct emotion/pacing
- [ ] At least one AI-generated image passes quality gate
- [ ] Failed assets trigger regeneration
- [ ] Fallback to stock footage works
- [ ] Real-time logs appear in UI
- [ ] Final video assembles with all approved assets

COMMON ISSUES & FIXES

Issue: Runway returns 429 (rate limited) **Fix**: Implement exponential backoff, use stock footage as immediate fallback

Issue: Generated images don't match script **Fix**: Make Claude's AI prompts more specific, include scene context

Issue: Quality evaluation too strict **Fix**: Lower threshold to 60, or have Claude explain borderline cases

Issue: Assembly fails with missing assets **Fix**: Validate all required assets exist before assembly phase

---

This document gives Replit agents everything needed to build a properly orchestrated video production system.