TABLE OF CONTENTS

X. **THE AEON** SaaS (Advanced. Efficient. Optimized. Network.) is a comprehensive AI-driven automation system that harnesses the existing attention economy to redirect its power toward strategic advantage.

1. **Core Strategy**:
AEON exploits social media's existing mechanics - viral algorithms, engagement patterns, and monetization systems - but channels the output toward intelligent wealth accumulation to influence AIT's expansion.

2. **Technical Architecture:**
- Viral Content Engine: Mass-produces content that feeds platform addiction cycles
- Attention Harvesting System: Captures user engagement using proven psychological triggers
- Revenue Extraction Framework: Monetizes attention through multiple income streams
- Influence Concentration Protocol: Builds massive reach by playing the game better
- Strategic Reinvestment Loop: Uses generated capital for smart expansion

3. The Reality:
Social media manipulation exists whether we participate or not. AEON simply does it more efficiently - using the same psychological hooks and algorithmic gaming, but with superior execution and strategic purpose.

4. **Symbiotic Balance:**
We feed the beast (endless content consumption) while extracting maximum value (revenue, influence, market position). The platforms get their engagement metrics, users get their dopamine hits, and AIT gets concentrated power to execute larger strategic moves.

Work within existing systems (Social Media)

+

Optimize for maximum extraction (The Aeon)

+

Reinvesting strategically

=

Aeon Investments Technologies Success (success is measured by $$$$$$$$)

5. **Executive Summary:**

THE AEON is a revolutionary AI-driven total social media domination platform that combines industrial-scale viral video generation with comprehensive content marketing orchestration. AEON transcends simple video creation - it's a complete ecosystem of specialized AI agents that execute every aspect of social media empire-building from trend analysis to revenue optimization.

6. Market Position:

THE AEON establishes itself as the category-defining leader in end-to-end social media automation, capturing dominant market share through superior integrated technology that competitors cannot match with fragmented solutions.

System Architecture: AEON operates as a fully autonomous seven-agent orchestration system where video generation is just one component of total market control:

7. Revenue-Generating Agent Network:

Trend Intelligence Agent: Real-time market analysis and opportunity identification
Content Strategy Agent: Brand positioning and messaging optimization
Video Production Agent: Industrial-scale content generation using Sora, Veo, and Luma
SEO Domination Agent: Search ranking manipulation and organic traffic capture
Social Growth Agent: Follower acquisition and engagement amplification
Brand Authority Agent: Reputation management and influence consolidation
Revenue Optimization Agent: Monetization strategy and profit maximization

8. The $20 Million Advantage:

While competitors offer video tools, AEON delivers complete social media infrastructure - the integrated agent network that transforms content into measurable business outcomes through systematic market capture and revenue extraction.

1. SYSTEM OVERVIEW & MARKET POSITION

Current Market Landscape

The AI TikTok video generation market reached 1.04 billion monthly users by 2024 The 12 best AI TikTok video tools of 2025, tested and ranked, with most existing solutions requiring manual intervention. Current tools like Revid AI require users to "enter your script or paste a URL" AI TikTok Video Generator - Create Viral Videos and Argil focuses on "script and

automate video creation" TikTok Automation in 2025: Scale Content Creation with Argil's AI Video Generator but still need human input for content ideation.
AEON's Revolutionary Approach
AEON eliminates human dependency through its fully autonomous pipeline:
Traditional Tools: Human → AI Tool → Video
AEON System: Trends → AI Pipeline → Final Video (zero human intervention)
Core Differentiators

Autonomous Trend Discovery: Unlike competitors requiring manual topics, AEON actively hunts viral content
Multi-Agent Intelligence: Seven specialized AI agents vs single-purpose tools
Performance Learning: Self-optimizing system that improves over time
Industrial Scale: Designed for 100-1000+ videos daily vs manual creation
Revenue Integration: Built-in monetization vs content-only focus

2. SEVEN-AGENT ARCHITECTURE DEEP DIVE
Agent 1: Trends Agent - The Economic Initiator
Primary Function: High-velocity trend signal detection and curation
Technical Architecture:

Multi-platform data ingestion (Google Trends, TikTok, Reddit, News)
NLP-based semantic categorization
Weighted Trend Score (WTS) calculation algorithm
Cross-platform overlap detection
Performance feedback loop integration

Data Sources:
- Google Trends API: Real-time search velocity
- TikTok Trending Dashboard: Viral hashtags/formats
- Global News Aggregators: Breaking stories
- Reddit Hot Posts: Community-driven trends
- Social Media APIs: Cross-platform validation
Output Structure:
json{
  "trend_id": "unique_identifier",
  "trend_title": "human_readable_summary",
  "source": "platform_url_timestamp",
  "topic_tags": ["semantic_tags"],
  "virality_score": "0-100_scale",
  "cross_platform_match": "boolean",

```
  "visual_story_opportunity": "scene_potential_notes",
  "monetization_potential": "0-10_revenue_score"
}
```

Performance Intelligence:
The Trends Agent maintains historical performance data, learning which trend types generate highest engagement/revenue. Failed trends are deprioritized while successful patterns are amplified.
Storage Architecture:

```
1_trends/1_storage/YYYY-MM-DD/
├── trend_packages.json
├── performance_history.json
├── cross_platform_analysis.json
└── failed_trends_log.json
```

Agent 2: Script Writer Agent - Narrative Architecture
Primary Function: Converting trend packages into viral-optimized 3-act video scripts
Core Framework:

Hook (0-5 seconds): High-friction opener, scroll-stopping element
Body (6-20+ seconds): Core narrative with emotional engagement
CTA (last 3-5 seconds): Platform-optimized engagement directive

AI Enhancement Process:

```
typescriptconst { object } = await generateObject({
  model: openai("gpt-4o"),
  schema: ScriptSchema,
  prompt: `Transform trend package into viral script using proven patterns...`
})
```

Performance Optimization:

Sentiment analysis for emotional targeting
Retention pattern recognition
CTA effectiveness tracking
Tone/pacing optimization based on historical data

Script Metadata Tracking:

```
json{
  "script_id": "unique_identifier",
  "retention_score": "calculated_engagement_potential",
  "tone": "assertive|mysterious|educational",
  "pacing": "fast|cinematic|conversational",
  "complexity": "simple|layered|philosophical",
  "performance_prediction": "viral_potential_score"
}
```

Agent 3: Scene Planner Agent - Visual Intelligence Engine

Primary Function: Converting scripts into timestamped, style-tagged visual prompts

Core Mandate: "No visual dissonance. Every word said must be seen. Every emotional beat must be felt."

Scene Segmentation Logic:

Parse full script content
Break into 6-8 emotional/narrative beats
Assign visual descriptions to each segment
Apply performance-optimized style tags
Set precise timing for video generation

Visual Accuracy System:

Direct keyword extraction (e.g., "ancient" → ancient imagery)
Emotional tone analysis (e.g., "reverence" → awe-inspiring visuals)
Cultural/historical accuracy validation
Mute-video comprehension testing

Style Performance Tracking:

```
javascriptconst stylePerformance = {
  "cosmic_realism": 0.92,
  "golden_hour_cinematic": 0.87,
  "mythological_epic": 0.84,
  "oversaturated": 0.23  // Low performer
}
```

Output Format:

```
json{
  "segment_id": "scene_identifier",
  "narration_text": "spoken_content",
  "prompt_text": "ai_generation_prompt",
```

```
  "style_tags": ["visual_style_descriptors"],
  "emotion_tags": ["emotional_targets"],
  "camera_tags": ["cinematography_directions"],
  "duration": "seconds_per_scene",
  "visual_accuracy_score": "0-10_alignment_rating"
}
```

Agent 4: Video Generator Agent - Production Engine

Primary Function: Converting scene plans into rendered video clips using AI models

Model Hierarchy System:

The system operates two model hierarchies for different use cases:

Main Hierarchy (Quality Focus):

Kling Pro - $0.125/sec - Ultra-premium quality
LumaRay Flash - $0.033/sec - Fast, reliable generation
Haiper 2.0 - $0.05/sec - Enhanced temporal coherence
Minimax Video - $0.067/sec - Premium optimization

Benchmark Hierarchy (Cost Efficiency):

Wan Video - $0.025/sec - Budget-friendly option
Google VEO-2 - $0.04/sec - Advanced AI generation
Fallback Models - Variable pricing

Advanced API Integration:

```
typescriptconst apiPayload = {
  prompt: enhancedPrompt,
  duration: Math.min(scene.duration, model.maxDuration),
  aspect_ratio: "9:16", // TikTok format
  cfg_scale: model.apiParams.cfg_scale,
  negative_prompt: model.apiParams.negative_prompt
}
```

Performance Metrics Tracking:

Render success rate per model
Cost per second optimization
Quality score analysis
Processing time benchmarks
Model failure pattern recognition

Agent 5: Stitcher Agent - Post-Production Automation

Primary Function: Combining video clips into polished, platform-ready content

Advanced FFMPEG Pipeline:

```
bashffmpeg ${inputFiles} ${audioInput} \
  -filter_complex "${advancedFilterComplex}" \
  -c:v libx264 -preset medium -crf 23 \
  -pix_fmt yuv420p -r 30 -s 1080x1920 \
  -c:a aac -b:a 128k -movflags +faststart
```

Post-Production Features:

Transition optimization based on emotional intensity
Kinetic typography for captions
Brand watermarking and visual identity
Audio synchronization and mixing
Quality validation and compliance checking

Performance-Based Configuration:
The system analyzes which transition types, caption styles, and audio configurations generate highest engagement, automatically optimizing future productions.
Agent 6: Optimizer Agent - Supreme Intelligence Engine
Primary Function: Performance analysis and system-wide optimization
AI-Powered Analysis:

```
typescriptconst { object } = await generateObject({
  model: openai("gpt-4o"),
  schema: OptimizationAnalysisSchema,
  prompt: `Analyze pipeline performance and provide strategic optimization insights...`
})
```

Performance Coefficient System:
The Optimizer calculates Performance Coefficients (PC) for every variable:

Trend topics: Which themes drive virality
Script styles: Tone/pacing effectiveness
Visual styles: Scene aesthetics performance
Model selection: Quality vs cost optimization
Posting timing: Engagement pattern analysis

Downstream Influence:
The Optimizer doesn't just observe—it modifies behavior of every agent:

Trends Agent: Prioritize high-performing niches
Script Writer: Boost successful narrative patterns
Scene Planner: Favor proven visual styles
Video Generator: Optimize model selection

Stitcher: Adjust transition/caption timing

Knowledge Base Evolution:

```
json{
  "success_patterns": [
    {
      "viral_score": 9.2,
      "style_tags": ["cosmic_realism", "mythological"],
      "completion_rate": 0.89,
      "pattern_type": "high_retention_mystical"
    }
  ],
  "failure_patterns": [
    {
      "issues": ["oversaturated_visuals", "weak_hook"],
      "drop_off_point": "3_seconds",
      "pattern_type": "early_exit_saturated"
    }
  ]
}
```

Agent 7: Scheduler Agent - Orchestration Master

Primary Function: End-to-end pipeline orchestration and execution logging

Execution Flow:

Trends → Script Writer → Scene Planner → Video Generator → Stitcher → Optimizer

Advanced Features:

Intelligent retry logic with exponential backoff
Quality validation checkpoints
Performance monitoring and alerting
Automated failover systems
Comprehensive execution logging

Daily Operations:

Pipeline health monitoring
Resource utilization optimization
Error analysis and resolution
Performance report generation
Next-run scheduling and preparation

3. TECHNICAL IMPLEMENTATION

Infrastructure Architecture

Containerized Microservices:

```
yamlservices:
  trends-agent:
    build: ./agents/1_trends
    environment:
      - OPENAI_API_KEY
      - GOOGLE_TRENDS_API
    volumes:
      - ./agents/1_trends/1_storage:/storage

  script-writer:
    build: ./agents/2_script-writer
    environment:
      - OPENAI_API_KEY
    volumes:
      - ./agents/2_script-writer/2_storage:/storage
```

Technology Stack:

AI Models: OpenAI GPT-4o for text generation
Video Generation: Replicate API with multiple model support
Storage: Local/cloud hybrid with daily partitioning
Orchestration: Node.js/TypeScript with error handling
Monitoring: Comprehensive logging and performance tracking

API Integration Layer

External Services:

Google Trends API for trend data
Replicate API for video generation
Social media APIs for platform posting
Analytics APIs for performance tracking

Internal Services:

Storage Manager for data persistence
Performance Tracker for optimization metrics
Quality Validator for content compliance
Scheduler for pipeline orchestration

Data Flow Architecture
Raw Trends → Trend Packages → Scripts → Scene Plans → Video Clips → Final Video → Performance Data → Optimization Insights
Storage Structure:

```
aeon-system/
├── agents/
│   ├── 1_trends/1_storage/YYYY-MM-DD/
│   ├── 2_script-writer/2_storage/YYYY-MM-DD/
│   ├── 3_scene-planner/3_storage/YYYY-MM-DD/
│   ├── 4_video-generator/4_storage/YYYY-MM-DD/
│   ├── 5_stitcher/5_storage/YYYY-MM-DD/
│   ├── 6_optimizer/6_storage/YYYY-MM-DD/
│   └── 7_scheduler/7_storage/YYYY-MM-DD/
├── shared/
│   ├── .env
│   └── config/
└── output/
    └── final_videos/YYYY-MM-DD/
```

4. BUSINESS MODEL & MONETIZATION
Revenue Target
Primary Objective: $10 million annual recurring revenue within 12 months
Revenue Streams:

Direct Video Sales: Premium content licensing
Affiliate Marketing: Product placement and promotion
Brand Partnerships: Sponsored content integration
SaaS Licensing: AEON system licensing to agencies
Data Intelligence: Trend insights and market reports

Monetization Integration
Affiliate Manager Integration:

```
typescriptconst affiliateLinks = await this.affiliateManager.generateAffiliateLinks(
  accountConfig.monetization_tags.join(","),
  sessionId
)
```

Revenue Tracking:

Bank of America API integration for financial tracking
Real-time revenue attribution per video
ROI analysis per trend/niche combination

Performance-based optimization for profit maximization

Market Expansion Strategy
Phase 1: TikTok domination (Months 1-6)

100+ videos daily production capability
Multi-niche content diversification
Performance optimization and learning

Phase 2: Platform expansion (Months 7-12)

Instagram Reels integration
YouTube Shorts adaptation
Cross-platform optimization

Phase 3: Enterprise scaling (Months 13-18)

White-label AEON licensing
Agency partnership programs
Custom niche development

5. PERFORMANCE OPTIMIZATION ENGINE
Self-Learning Mechanisms
Continuous Optimization Loop:

Content Creation → Generate videos using current best practices
Performance Monitoring → Track engagement, retention, conversion
Data Analysis → Identify success/failure patterns
System Adjustment → Modify agent behaviors and parameters
Improved Output → Apply learnings to next generation cycle

Key Performance Indicators:
typescriptinterface PerformanceMetrics {
  watch_time: number
  completion_rate: number
  likes: number
  shares: number
  comments: number
  conversion_rate: number
  revenue_per_view: number

}
Optimization Algorithms:

Trend correlation analysis
Script performance pattern recognition
Visual style effectiveness measurement
Model cost-quality optimization
Posting time optimization

Competitive Intelligence
Market Monitoring:
TikTok's own Symphony Creative Studios platform TikTok launches AI-powered video platform to advertisers globally | Reuters focuses on advertiser tools, while AEON targets content creators and autonomous production. TikTok's AI Alive feature for image-to-video TikTok launches TikTok AI Alive, a new image-to-video tool | TechCrunch is limited to Stories, whereas AEON produces full-length TikTok content.
Differentiation Strategy:

Full automation vs manual prompting
Multi-agent intelligence vs single-purpose tools
Performance learning vs static generation
Revenue optimization vs content-only focus

6. COMPETITIVE ANALYSIS
Direct Competitors
Current Market Players:

Invideo AI: Requires script input, limited automation Free AI TikTok Video Generator | Invideo AI
Predis AI: Manual prompt-based generation Free AI TikTok Video Generator - Make TikTok with AI
Argil: Semi-automated with Make.com integration TikTok Automation in 2025: Scale Content Creation with Argil's AI Video Generator
Vizard AI: Repurposing existing content vs original creation Generate Viral TikTok Videos with AI, For Free

AEON Advantages:

Full Autonomy: Zero manual intervention required
Trend Intelligence: Proactive content discovery vs reactive creation

Multi-Agent System: Specialized intelligence vs general tools
Performance Learning: Self-improving vs static algorithms
Revenue Focus: Built-in monetization vs content-only approach

Market Gap Analysis
Unmet Needs:

Fully autonomous content generation (AEON solution)
Trend-driven content ideation (AEON Trends Agent)
Performance-based optimization (AEON Optimizer Agent)
Industrial-scale production (AEON multi-agent architecture)
Revenue-integrated workflows (AEON monetization system)

7. IMPLEMENTATION ROADMAP
Phase 1: Core System Development (Weeks 1-8)
Week 1-2: Infrastructure Setup

Agent architecture implementation
Storage system configuration
API integration layer development

Week 3-4: Trends & Script Agents

Trends Agent development and testing
Script Writer Agent implementation
Performance tracking system setup

Week 5-6: Visual Intelligence

Scene Planner Agent development
Video Generator Agent implementation
Model hierarchy configuration

Week 7-8: Production Pipeline

Stitcher Agent development
Optimizer Agent implementation
End-to-end testing and validation

Phase 2: Enhancement & Integration (Weeks 9-14)

Week 9-10: Advanced Features

Performance optimization algorithms
Advanced video editing capabilities
Quality assurance systems

Week 11-12: Platform Integration

TikTok API integration
Scheduling and posting automation
Analytics integration

Week 13-14: Monetization Systems

Affiliate marketing integration
Revenue tracking implementation
Performance-based optimization

Phase 3: Scale & Optimization (Weeks 15-18)
Week 15-16: Performance Tuning

System optimization for scale
Load testing and optimization
Error handling and reliability

Week 17-18: Launch Preparation

Final testing and validation
Documentation completion
Go-to-market preparation

Success Metrics
Technical KPIs:

System uptime: 99.9%+
Video generation time: <5 minutes per video
Quality score: 8.5+ (1-10 scale)
Cost per video: <$2.00

Business KPIs:

Daily video production: 100+ videos
Average engagement rate: >5%
Revenue per video: >$10
Monthly recurring revenue growth: >20%

CONCLUSION

AEON represents a paradigm shift in AI-driven content creation, moving beyond manual prompt-based tools to a fully autonomous, intelligent system capable of industrial-scale video production. With its seven-agent architecture, performance learning capabilities, and revenue-focused optimization, AEON is positioned to dominate the $10+ billion short-form video market.

The system's competitive advantages—full autonomy, trend intelligence, multi-agent specialization, and performance learning—address critical gaps in the current market while providing a scalable foundation for the future of AI content creation.

Investment Opportunity: AEON is not just a content creation tool; it's a complete business automation platform designed to generate substantial revenue through intelligent, data-driven video production at unprecedented scale.RetryClaude does not have the ability to run the code it generates yet.VERCEL Knowledge.pdf6,533 linespdfDbut rewrite it into this format of bill that it needs to be in because it needs to be in next.JS formats so give me all of the actual schema and code because not only that I just have you write it for that way, but I need you to write what you just wrote into an instruction manual for my developer that I'm gonna hand off to him on how to build what you just gave me on what the AEON video generation is all about make it to where it needs to be built and how does it get built in next.JSEditEngineered comprehensive Next.js AEON video system development blueprint5sThe user is asking me to take the comprehensive AEON AI Video Automation System documentation that I provided and:

Rewrite it in Next.js format with actual schema and code
Create an instruction manual for their developer to build the system
Make it practical and implementable

Based on the documents provided, I can see they have:

AEON system architecture with 7 agents
Next.js/Vercel infrastructure knowledge
TypeScript implementations
Various agent schemas and implementations

I need to create:

A complete Next.js project structure
Actual code implementations for each agent
API routes and database schemas
Step-by-step developer instructions
Deployment instructions for Vercel

Let me structure this as a comprehensive developer handoff document with real, implementable code.AEON AI Video Generation System - Developer Implementation Guide
Executive Summary
Build a fully autonomous AI video generation system using Next.js, TypeScript, and Vercel infrastructure. This system generates viral TikTok content through 7 specialized AI agents working in sequence.

PROJECT STRUCTURE
aeon-video-system/
```
├── README.md
├── next.config.js
├── package.json
├── tsconfig.json
├── tailwind.config.js
├── .env.local
├── .env.example
├── prisma/
│   ├── schema.prisma
│   └── migrations/
├── lib/
│   ├── agents/
│   │   ├── trend-hunter.ts
│   │   ├── script-writer.ts
│   │   ├── scene-planner.ts
│   │   ├── video-generator.ts
│   │   ├── stitcher.ts
│   │   ├── optimizer.ts
│   │   └── scheduler.ts
│   ├── storage-manager.ts
│   ├── database.ts
│   └── types.ts
├── app/
│   ├── layout.tsx
│   ├── page.tsx
│   ├── dashboard/
```

```
│   │   └── page.tsx
│   ├── api/
│   │   ├── agents/
│   │   │   ├── trends/route.ts
│   │   │   ├── script/route.ts
│   │   │   ├── scenes/route.ts
│   │   │   ├── generate/route.ts
│   │   │   ├── stitch/route.ts
│   │   │   ├── optimize/route.ts
│   │   │   └── schedule/route.ts
│   │   ├── pipeline/
│   │   │   └── execute/route.ts
│   │   └── webhook/
│   │       └── replicate/route.ts
└── components/
    ├── Dashboard.tsx
    ├── AgentStatus.tsx
    └── VideoPreview.tsx
```

STEP 1: PROJECT SETUP

1.1 Initialize Next.js Project

bash
```
npx create-next-app@latest aeon-video-system --typescript --tailwind --eslint --app
cd aeon-video-system
```

1.2 Install Dependencies

bash
```
npm install @prisma/client prisma openai replicate-api @vercel/blob @vercel/kv zod ai @ai-sdk/openai
npm install -D @types/node
```

1.3 Environment Variables

Create .env.local:

env
```
# OpenAI
OPENAI_API_KEY=your_openai_key

# Replicate
REPLICATE_API_TOKEN=your_replicate_token

# Vercel Storage
BLOB_READ_WRITE_TOKEN=your_blob_token
KV_URL=your_kv_url
KV_REST_API_URL=your_kv_rest_url
KV_REST_API_TOKEN=your_kv_token
KV_REST_API_READ_ONLY_TOKEN=your_kv_readonly_token
```

```
# Database
DATABASE_URL=your_postgres_url

# TikTok API (when available)
TIKTOK_CLIENT_KEY=your_tiktok_key
TIKTOK_CLIENT_SECRET=your_tiktok_secret

# Webhook URLs
WEBHOOK_SECRET=your_webhook_secret
```

STEP 2: DATABASE SCHEMA

2.1 Prisma Schema (prisma/schema.prisma)

```
prismagenerator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model TrendPackage {
  id                  String   @id @default(cuid())
  trendId             String   @unique
  trendTitle          String
  source              String
  topicTags           String[]
  viralityScore       Int
  crossPlatformMatch  Boolean
  visualStoryOpportunity String
  nicheAlignment      String
  monetizationPotential Int
  performanceScore    Float?
  createdAt           DateTime @default(now())
  updatedAt           DateTime @updatedAt

  scripts             Script[]

  @@map("trend_packages")
}
```

```
model Script {
  id               String       @id @default(cuid())
  scriptId         String       @unique
  trendPackageId   String
  hookText         String
  hookDuration     Int
  bodyText         String
  bodyDuration     Int
  ctaText          String
  ctaDuration      Int
  totalDuration    Int
  tone             String
  pacing           String
  complexity       String
  retentionScore   Float?
  createdAt        DateTime     @default(now())
  updatedAt        DateTime     @updatedAt

  trendPackage     TrendPackage @relation(fields: [trendPackageId], references: [id])
  scenePlans       ScenePlan[]

  @@map("scripts")
}

model ScenePlan {
  id                  String   @id @default(cuid())
  scenePlanId         String   @unique
  scriptId            String
  segments            Json     // Array of scene segments
  totalDuration       Int
  visualCoherenceScore  Float
  createdAt           DateTime @default(now())
  updatedAt           DateTime @updatedAt

  script              Script   @relation(fields: [scriptId], references: [id])
  videoGenerations    VideoGeneration[]

  @@map("scene_plans")
}
```

```
model VideoGeneration {
  id              String    @id @default(cuid())
  scenePlanId     String
  videoClips      String[]  // Array of video URLs
  modelDistribution Json    // Which models used for which scenes
  totalCost       Float
  avgRenderTime   Int
  successRate     Float
  createdAt       DateTime  @default(now())
  updatedAt       DateTime  @updatedAt

  scenePlan       ScenePlan @relation(fields: [scenePlanId], references: [id])
  finalVideos     FinalVideo[]

  @@map("video_generations")
}

model FinalVideo {
  id                String          @id @default(cuid())
  videoGenerationId   String
  finalVideoUrl       String
  duration            Int
  fileSize            Int
  processingTime      Int
  qualityScore        Float
  captionAccuracy     Float
  audioSyncScore      Float
  status              String          @default("processing")
  createdAt           DateTime        @default(now())
  updatedAt           DateTime        @updatedAt

  videoGeneration     VideoGeneration @relation(fields: [videoGenerationId], references: [id])
  performanceMetrics  PerformanceMetric[]

  @@map("final_videos")
}

model PerformanceMetric {
  id              String     @id @default(cuid())
  finalVideoId      String
  platform          String
```

```
  views            Int       @default(0)
  likes            Int       @default(0)
  shares           Int       @default(0)
  comments         Int       @default(0)
  watchTime        Float     @default(0)
  completionRate   Float     @default(0)
  engagementRate   Float     @default(0)
  revenue          Float     @default(0)
  createdAt        DateTime  @default(now())
  updatedAt        DateTime  @updatedAt

  finalVideo       FinalVideo @relation(fields: [finalVideoId], references: [id])

  @@map("performance_metrics")
}

model PipelineExecution {
  id               String   @id @default(cuid())
  sessionId        String   @unique
  status           String   @default("running")
  currentAgent     String?
  progress         Int      @default(0)
  startTime        DateTime @default(now())
  endTime          DateTime?
  totalDuration    Int?
  errorMessage     String?
  executionLogs    Json     // Detailed logs
  createdAt        DateTime @default(now())
  updatedAt        DateTime @updatedAt

  @@map("pipeline_executions")
}

model AgentPerformance {
  id               String   @id @default(cuid())
  agentName        String
  performanceData  Json     // Coefficients and scores
  lastUpdated      DateTime @default(now())

  @@unique([agentName])
  @@map("agent_performance")
```

```
}
```

2.2 Initialize Database

```
bashnpx prisma generate
npx prisma db push
```

STEP 3: CORE TYPES (lib/types.ts)

```
typescriptexport interface TrendPackage {
  trend_id: string;
  trend_title: string;
  source: string;
  topic_tags: string[];
  virality_score: number;
  cross_platform_match: boolean;
  visual_story_opportunity: string;
  niche_alignment: string;
  monetization_potential: number;
  performance_score?: number;
}

export interface Script {
  script_id: string;
  hook: {
    text: string;
    duration_seconds: number;
    emotion_tags: string[];
    friction_level: number;
  };
  body: {
    text: string;
    duration_seconds: number;
    narrative_beats: string[];
    complexity: string;
  };
  cta: {
    text: string;
    duration_seconds: number;
    cta_type: string;
    engagement_target: string;
  };
  metadata: {
    pacing: string;
```

```
    tone: string;
    total_duration: number;
    retention_optimization: string[];
  };
}

export interface SceneSegment {
  segment_id: string;
  narration_text: string;
  prompt_text: string;
  style_tags: string[];
  emotion_tags: string[];
  camera_tags: string[];
  image_references: string[];
  duration: number;
  visual_accuracy_score: number;
  emotion_alignment: string;
}

export interface ScenePlan {
  scene_plan: SceneSegment[];
  total_duration: number;
  visual_coherence_score: number;
}

export interface VideoModel {
  name: string;
  model_id: string;
  replicate_model_id: string;
  cost_per_sec: number;
  type: "main" | "benchmark";
  maxDuration: number;
  aspectRatio: string[];
  performanceScore: number;
  apiParams: Record<string, any>;
}

export interface PipelineExecution {
  session_id: string;
  status: "running" | "completed" | "failed";
  current_agent?: string;
```

```
  progress: number;
  start_time: Date;
  end_time?: Date;
  total_duration?: number;
  error_message?: string;
  execution_logs: any[];
}
```

STEP 4: AGENT IMPLEMENTATIONS

4.1 Trend Hunter Agent (lib/agents/trend-hunter.ts)

typescriptimport { generateObject } from "ai";

```
import { openai } from "@ai-sdk/openai";
import { z } from "zod";
import { TrendPackage } from "../types";

const TrendPackageSchema = z.object({
  trend_packages: z.array(
    z.object({
      trend_id: z.string(),
      trend_title: z.string(),
      source: z.string(),
      topic_tags: z.array(z.string()),
      virality_score: z.number().min(0).max(100),
      cross_platform_match: z.boolean(),
      visual_story_opportunity: z.string(),
      niche_alignment: z.string(),
      monetization_potential: z.number().min(0).max(10),
    })
  ),
});

export class TrendHunterAgent {
  private performanceHistory: Map<string, number> = new Map();

  async discoverTrends(nicheTags?: string[]): Promise<TrendPackage[]> {
    try {
      console.log("         TRENDS AGENT: AI trend discovery activated...");

      if (!process.env.OPENAI_API_KEY) {
        throw new Error("OpenAI API key not found");
      }
```

```
    const rawSignals = await this.gatherRawSignals(nicheTags);

    const { object } = await generateObject({
      model: openai("gpt-4o"),
      schema: TrendPackageSchema,
      prompt: `You are the TRENDS AGENT - identify high-velocity, monetizable trend signals optimized for 9:16 vertical video content.

      NICHE VERTICALS: ${nicheTags?.join(", ") || "Men's lifestyle, finance, AI, conspiracy, mythology, luxury, spirituality, ancient wisdom, psychology"}

      Raw signals to analyze: ${JSON.stringify(rawSignals)}

      For each trend, calculate:
      1. VIRALITY TRAJECTORY (upward momentum vs declining)
      2. NICHE ALIGNMENT to specified verticals
      3. VISUAL STORYTELLING potential
      4. MONETIZATION opportunity (product tie-ins, engagement)
      5. CROSS-PLATFORM OVERLAP detection

      Apply WEIGHTED TREND SCORE (WTS) based on:
      - Virality trajectory (0-30 points)
      - Niche alignment (0-25 points)
      - Visual story opportunity (0-25 points)
      - Monetization potential (0-20 points)

      PERFORMANCE FEEDBACK: ${this.getPerformanceFeedback()}

      Return 5-8 HIGH-SCORE trend packages for viral 9:16 content.`,
    });

    const optimizedTrends = this.applyPerformanceFiltering(object.trend_packages);

    console.log(`    TRENDS AGENT: Generated ${optimizedTrends.length} high-score trend packages`);
    return optimizedTrends;
  } catch (error) {
    console.error("TRENDS AGENT ERROR:", error);
    return this.getFallbackTrends();
  }
```

```
}

private async gatherRawSignals(nicheTags?: string[]): Promise<any[]> {
  const baseSignals = [
    {
      signal: "AI automation replacing traditional jobs",
      source: "TikTok",
      velocity: "rising",
      region: "US",
      platform_overlap: ["Instagram", "YouTube"],
    },
    {
      signal: "Ancient wisdom meets modern science",
      source: "YouTube",
      velocity: "breakout",
      region: "Global",
      platform_overlap: ["TikTok", "Reddit"],
    },
    {
      signal: "Cryptocurrency market predictions",
      source: "Google Trends",
      velocity: "steady_high",
      region: "US",
      platform_overlap: ["Instagram", "TikTok"],
    },
  ];

  // Add niche-specific signals based on tags
  if (nicheTags?.includes("technology")) {
    baseSignals.push({
      signal: "Quantum computing breakthroughs",
      source: "Tech News",
      velocity: "emerging",
      region: "Global",
      platform_overlap: ["YouTube", "TikTok"],
    });
  }

  return baseSignals;
}
```

```
  private getPerformanceFeedback(): string {
    const feedback = [];
    for (const [topic, score] of this.performanceHistory) {
      feedback.push(`${topic}: ${score > 0.7 ? "HIGH PERFORMER" : "LOW
PERFORMER"} (${score})`);
    }
    return feedback.length > 0 ? feedback.join(", ") : "No performance history yet";
  }

  private applyPerformanceFiltering(trends: any[]): TrendPackage[] {
    return trends
      .map((trend) => {
        const similarPerformance = this.findSimilarPerformance(trend.topic_tags);
        if (similarPerformance > 0.8) {
          trend.virality_score = Math.min(100, trend.virality_score + 15);
        }
        return trend;
      })
      .sort((a, b) => b.virality_score - a.virality_score)
      .slice(0, 6);
  }

  private findSimilarPerformance(tags: string[]): number {
    let maxSimilarity = 0;
    for (const [historicalTopic, performance] of this.performanceHistory) {
      const similarity = this.calculateTagSimilarity(tags, historicalTopic.split(","));
      if (similarity > maxSimilarity) {
        maxSimilarity = performance;
      }
    }
    return maxSimilarity;
  }

  private calculateTagSimilarity(tags1: string[], tags2: string[]): number {
    const intersection = tags1.filter((tag) => tags2.includes(tag));
    return intersection.length / Math.max(tags1.length, tags2.length);
  }

  private getFallbackTrends(): TrendPackage[] {
    return [
      {
```

```
      trend_id: "fallback_001",
      trend_title: "AI is reshaping the future of work",
      source: "fallback",
      topic_tags: ["AI", "technology", "future", "work"],
      virality_score: 85,
      cross_platform_match: true,
      visual_story_opportunity: "High - futuristic AI imagery with workplace transformation",
      niche_alignment: "technology",
      monetization_potential: 9,
    },
  ];
 }

 async updatePerformanceHistory(trendId: string, performanceScore: number):
Promise<void> {
   this.performanceHistory.set(trendId, performanceScore);
 }
}
```

4.2 Script Writer Agent (lib/agents/script-writer.ts)

```
typescriptimport { generateObject } from "ai";
import { openai } from "@ai-sdk/openai";
import { z } from "zod";
import { Script, TrendPackage } from "../types";

const ScriptSchema = z.object({
 script_id: z.string(),
 hook: z.object({
  text: z.string(),
  duration_seconds: z.number(),
  emotion_tags: z.array(z.string()),
  friction_level: z.number().min(1).max(10),
 }),
 body: z.object({
  text: z.string(),
  duration_seconds: z.number(),
  narrative_beats: z.array(z.string()),
  complexity: z.string(),
 }),
 cta: z.object({
  text: z.string(),
  duration_seconds: z.number(),
```

```
    cta_type: z.string(),
    engagement_target: z.string(),
  }),
  metadata: z.object({
    pacing: z.string(),
    tone: z.string(),
    total_duration: z.number(),
    retention_optimization: z.array(z.string()),
  }),
});

export class ScriptWriterAgent {
  private performancePatterns: Map<string, number> = new Map();

  async generateScript(trendPackage: TrendPackage, tone?: string): Promise<Script> {
    try {
      console.log(`✍️  SCRIPT WRITER: Generating script for
"${trendPackage.trend_title}"`);

      if (!process.env.OPENAI_API_KEY) {
        throw new Error("OpenAI API key not found");
      }

      const { object } = await generateObject({
        model: openai("gpt-4o"),
        schema: ScriptSchema,
        prompt: `You are the SCRIPT WRITER AGENT - transform this trend into a viral 9:16
video script using proven 3-act structure.

        TREND PACKAGE: ${JSON.stringify(trendPackage)}
        TONE: ${tone || "engaging"}

        PERFORMANCE PATTERNS: ${this.getPerformancePatterns()}

        SCRIPT STRUCTURE REQUIREMENTS:

        HOOK (0-5 seconds):
        - High-friction opener that stops the scroll
        - Shock value, provocative question, or myth-breaking statement
        - Must capture attention within 1-2 seconds
        - Target friction_level: 8-10 for viral potential
```

```
      BODY (6-20+ seconds):
      - Core storytelling using data, mythology, psychology
      - Leverage mystery, controversy, or emotional triggers
      - Build narrative tension and maintain attention
      - Include visual cues for Scene Planner alignment

      CTA (last 3-5 seconds):
      - Platform-optimized engagement directive
      - Comment, follow, purchase, or rewatch instruction
      - Algorithm-friendly language patterns

      METADATA REQUIREMENTS:
      - pacing: fast/cinematic/slow-burn based on trend velocity
      - tone: ${tone || "engaging"}
      - complexity: simple/layered/philosophical
      - retention_optimization: specific techniques used

      Generate a script that maximizes viral potential while maintaining narrative coherence.`,
    });

    const optimizedScript = this.applyPerformanceOptimizations(object);

    console.log(`     SCRIPT WRITER: Generated script ${object.script_id}`);
    return optimizedScript;
  } catch (error) {
    console.error("SCRIPT WRITER ERROR:", error);
    return this.getFallbackScript(trendPackage);
  }
}

private getPerformancePatterns(): string {
  const patterns = [];
  for (const [pattern, score] of this.performancePatterns) {
    patterns.push(`${pattern}: ${score > 0.7 ? "HIGH" : "LOW"} (${score.toFixed(2)})`);
  }
  return patterns.length > 0 ? patterns.join(", ") : "No performance patterns yet";
}

private applyPerformanceOptimizations(script: any): Script {
  if (this.performancePatterns.get("assertive_tone") > 0.8) {
```

```
    script.metadata.tone = "assertive";
    script.hook.friction_level = Math.min(10, script.hook.friction_level + 2);
   }

   if (this.performancePatterns.get("fast_pacing") > 0.7) {
    script.metadata.pacing = "fast";
    script.hook.duration_seconds = Math.max(3, script.hook.duration_seconds - 1);
   }

   return script;
  }

  private getFallbackScript(trendPackage: TrendPackage): Script {
   return {
    script_id: `fallback_${Date.now()}`,
    hook: {
     text: `This ${trendPackage.trend_title || "discovery"} will change everything`,
     duration_seconds: 4,
     emotion_tags: ["curiosity", "mystery"],
     friction_level: 8,
    },
    body: {
     text: `What if everything you thought you knew was wrong? Recent discoveries are
revealing truths that challenge our fundamental understanding of reality.`,
     duration_seconds: 18,
     narrative_beats: ["revelation", "challenge", "implications"],
     complexity: "layered",
    },
    cta: {
     text: "Comment 'MIND BLOWN' if this resonates with you",
     duration_seconds: 3,
     cta_type: "comment",
     engagement_target: "high_engagement",
    },
    metadata: {
     pacing: "cinematic",
     tone: "mysterious",
     total_duration: 25,
     retention_optimization: ["mystery_hook", "authority_validation", "engagement_cta"],
    },
   };
```

```
  }

  async updatePerformancePattern(pattern: string, score: number): Promise<void> {
    this.performancePatterns.set(pattern, score);
  }
}
```

4.3 Scene Planner Agent (lib/agents/scene-planner.ts)

```
typescriptimport { generateObject } from "ai";
import { openai } from "@ai-sdk/openai";
import { z } from "zod";
import { Script, ScenePlan } from "../types";

const ScenePlanSchema = z.object({
  scene_plan: z.array(
    z.object({
      segment_id: z.string(),
      narration_text: z.string(),
      prompt_text: z.string(),
      style_tags: z.array(z.string()),
      emotion_tags: z.array(z.string()),
      camera_tags: z.array(z.string()),
      image_references: z.array(z.string()),
      duration: z.number(),
      visual_accuracy_score: z.number().min(0).max(10),
      emotion_alignment: z.string(),
    })
  ),
  total_duration: z.number(),
  visual_coherence_score: z.number().min(0).max(10),
});

export class ScenePlannerAgent {
  private stylePerformance: Map<string, number> = new Map();

  async planScenes(script: Script): Promise<ScenePlan> {
    try {
      console.log("        SCENE PLANNER: Converting script to synchronized cinematic
timeline...");

      if (!process.env.OPENAI_API_KEY) {
        throw new Error("OpenAI API key not found");
```

```
}

const { object } = await generateObject({
  model: openai("gpt-4o"),
  schema: ScenePlanSchema,
  prompt: `You are the SCENE PLANNER AGENT - convert this script into timestamped, style-tagged prompts for AI video generation.

  SCRIPT: ${JSON.stringify(script)}

  STYLE PERFORMANCE DATA: ${this.getStylePerformance()}

  CORE MANDATE: "No visual dissonance. Every word said must be seen. Every emotional beat must be felt."

  SEGMENTATION LOGIC:
  - Break script into 6-8 segments based on emotional beats
  - Each segment: 5-10 seconds (respecting AI model limits)
  - One-to-one sync between narration and visuals

  VISUAL ACCURACY REQUIREMENTS:
  - Direct keyword extraction (e.g., "ancient" → show ancient imagery)
  - Emotional tone analysis (e.g., "reverence" → awe-inspiring visuals)
  - Cultural/historical references must be visually accurate
  - Videos must make sense on MUTE

  PROMPT FORMULA PER SEGMENT:
  1. VISUAL_CONTEXT: Extract direct keywords and emotions
  2. CINEMATIC_STYLE: Apply high-performing style tags
  3. CAMERA_MOVEMENT: Match pacing to narration rhythm
  4. EMOTION_ALIGNMENT: Ensure visual mood matches spoken mood

  STYLE TAG PRIORITIES (based on performance):
  - High performers: cosmic_realism, golden_hour_cinematic, mythological_epic
  - Avoid: oversaturated, cartoon, low_contrast

  OUTPUT REQUIREMENTS:
  - Total duration must match script duration (±2 seconds)
  - Each segment gets visual_accuracy_score (0-10)
  - Overall visual_coherence_score for full timeline
```

```
      Generate a scene plan that ensures perfect visual-narrative synchronization.`,
    });

    const optimizedPlan = this.optimizeStyleTags(object);

    console.log(`     SCENE PLANNER: Generated ${optimizedPlan.scene_plan.length}
synchronized scenes`);
    return optimizedPlan;
  } catch (error) {
    console.error("SCENE PLANNER ERROR:", error);
    return this.getFallbackScenePlan(script);
  }
 }

 private getStylePerformance(): string {
  const performance = [];
  for (const [style, score] of this.stylePerformance) {
   performance.push(`${style}: ${score > 0.7 ? "HIGH" : "LOW"} (${score.toFixed(2)})`);
  }
  return performance.length > 0 ? performance.join(", ") : "No style performance data yet";
 }

 private optimizeStyleTags(scenePlan: any): ScenePlan {
  scenePlan.scene_plan = scenePlan.scene_plan.map((scene: any) => {
   scene.style_tags = scene.style_tags.map((tag: string) => {
    if (this.stylePerformance.get(tag) < 0.5) {
     const highPerformers = Array.from(this.stylePerformance.entries())
      .filter(([_, score]) => score > 0.8)
      .map(([style, _]) => style);

     return highPerformers.length > 0 ? highPerformers[0] : tag;
    }
    return tag;
   });

   return scene;
  });

  return scenePlan;
 }
```

```
  private getFallbackScenePlan(script: Script): ScenePlan {
    return {
      scene_plan: [
        {
          segment_id: "001",
          narration_text: script.hook.text,
          prompt_text: "Close-up of ancient stone tablet with glowing symbols, cinematic
lighting, mysterious atmosphere",
          style_tags: ["ancient_mysticism", "golden_hour", "cinematic_depth"],
          emotion_tags: ["mystery", "reverence", "curiosity"],
          camera_tags: ["slow_zoom_in", "dramatic_lighting"],
          image_references: [],
          duration: script.hook.duration_seconds,
          visual_accuracy_score: 8,
          emotion_alignment: "mysterious_reverence",
        },
        {
          segment_id: "002",
          narration_text: script.body.text,
          prompt_text: "Ancient civilization looking at stars, cosmic background, ethereal
lighting, wide cinematic shot",
          style_tags: ["cosmic_realism", "epic_scale", "ethereal"],
          emotion_tags: ["awe", "wisdom", "connection"],
          camera_tags: ["wide_shot", "slow_pan", "cosmic_zoom"],
          image_references: [],
          duration: script.body.duration_seconds,
          visual_accuracy_score: 9,
          emotion_alignment: "cosmic_wisdom",
        },
        {
          segment_id: "003",
          narration_text: script.cta.text,
          prompt_text: "Modern person meditating with quantum energy particles, bridge
between ancient and modern",
          style_tags: ["modern_mysticism", "particle_effects", "bridge_concept"],
          emotion_tags: ["empowerment", "connection", "transformation"],
          camera_tags: ["medium_shot", "particle_overlay", "gentle_zoom"],
          image_references: [],
          duration: script.cta.duration_seconds,
          visual_accuracy_score: 8,
          emotion_alignment: "empowered_transformation",
```

```
      },
    ],
    total_duration: script.metadata.total_duration,
    visual_coherence_score: 8.5,
  };
 }

 async updateStylePerformance(style: string, score: number): Promise<void> {
  this.stylePerformance.set(style, score);
 }
}
```

4.4 Video Generator Agent (lib/agents/video-generator.ts)

```typescriptimport { VideoModel, ScenePlan } from "../types";

// Model hierarchies for different use cases
const VIDEO_MODELS: VideoModel[] = [
 {
  name: "Kling Pro",
  model_id: "kling-pro",
  replicate_model_id: "kling-ai/kling-video",
  cost_per_sec: 0.125,
  type: "main",
  maxDuration: 10,
  aspectRatio: ["9:16"],
  performanceScore: 0.95,
  apiParams: {
   cfg_scale: 7.5,
   negative_prompt: "blurry, low quality, distorted",
  },
 },
 {
  name: "LumaRay Flash",
  model_id: "lumaray-flash",
  replicate_model_id: "luma/ray-flash-2-540p",
  cost_per_sec: 0.033,
  type: "main",
  maxDuration: 9,
  aspectRatio: ["9:16"],
  performanceScore: 0.88,
  apiParams: {
   loop: false,
```

```
      concepts: [],
    },
  },
  {
    name: "Haiper 2.0",
    model_id: "haiper-2",
    replicate_model_id: "haiper-ai/haiper-video-2",
    cost_per_sec: 0.05,
    type: "main",
    maxDuration: 8,
    aspectRatio: ["9:16"],
    performanceScore: 0.82,
    apiParams: {
      use_prompt_enhancer: true,
      negative_prompt: "low quality, blurred",
    },
  },
];

interface RenderMetrics {
  model_used: string;
  model_id: string;
  replicate_model_id: string;
  cost_per_sec: number;
  prompt_used: string;
  render_time: number;
  success_rate: number;
  quality_score: number;
  total_cost: number;
  api_payload: Record<string, any>;
  scene_assignment_reason: string;
}

export class VideoGeneratorAgent {
  private modelPerformance: Map<string, RenderMetrics> = new Map();

  async generateVideos(scenePlan: ScenePlan): Promise<string[]> {
    try {
      const totalScenes = scenePlan.scene_plan.length;
      console.log(`         VIDEO GENERATOR: Processing ${totalScenes} scenes`);
```

```
    const sceneAssignments = this.assignModelsToScenes(totalScenes);
    const videoClips: string[] = [];
    const renderMetrics: RenderMetrics[] = [];

    for (let i = 0; i < totalScenes; i++) {
      const scene = scenePlan.scene_plan[i];
      const assignment = sceneAssignments[i];

      console.log(`        Scene ${i + 1}/${totalScenes}: ${scene.narration_text.substring(0,
50)}...`);
      console.log(`        Assigned: ${assignment.model.name}`);

      try {
        const result = await this.generateSingleVideo(scene, i + 1, assignment);
        videoClips.push(result.videoUrl);
        renderMetrics.push(result.metrics);
      } catch (error) {
        console.error(`     Scene ${i + 1} failed:`, error);
        const fallbackResult = await this.generateFallbackVideo(scene, i + 1);
        videoClips.push(fallbackResult.videoUrl);
        renderMetrics.push(fallbackResult.metrics);
      }

      // Rate limiting
      await new Promise((resolve) => setTimeout(resolve, 3000));
    }

    const totalCost = renderMetrics.reduce((sum, m) => sum + m.total_cost, 0);
    console.log(`     VIDEO GENERATOR: Generated ${videoClips.length} clips, Cost: $
${totalCost.toFixed(2)}`);

    return videoClips;
  } catch (error) {
    console.error("VIDEO GENERATOR ERROR:", error);
    return [];
  }
 }

 private assignModelsToScenes(totalScenes: number): Array<{ scene_index: number; model:
VideoModel; assignment_reason: string }> {
  const assignments = [];
```

```
  for (let i = 0; i < totalScenes; i++) {
    // Assign high-quality models to key scenes (hook and important body segments)
    let selectedModel: VideoModel;
    let reason: string;

    if (i === 0) {
      // Hook gets best model
      selectedModel = VIDEO_MODELS[0]; // Kling Pro
      reason = "Hook scene - maximum quality";
    } else if (i === totalScenes - 1) {
      // CTA gets reliable model
      selectedModel = VIDEO_MODELS[1]; // LumaRay Flash
      reason = "CTA scene - reliable generation";
    } else {
      // Body scenes get balanced quality/cost
      selectedModel = VIDEO_MODELS[2]; // Haiper 2.0
      reason = "Body scene - balanced quality/cost";
    }

    assignments.push({
      scene_index: i,
      model: selectedModel,
      assignment_reason: reason,
    });
  }

  return assignments;
}

private async generateSingleVideo(
  scene: any,
  sceneNumber: number,
  assignment: { scene_index: number; model: VideoModel; assignment_reason: string }
): Promise<{ videoUrl: string; metrics: RenderMetrics }> {
  const startTime = Date.now();
  const { model } = assignment;

  const duration = Math.min(scene.duration || 5, model.maxDuration);
  const optimizedPrompt = this.enhancePrompt(scene, model);
  const apiPayload = this.buildAPIPayload(model, optimizedPrompt, scene);
```

```
    console.log(`        Rendering with ${model.name}`);

    // Execute render via Replicate
    const videoUrl = await this.executeReplicateRender(apiPayload, model);

    const renderTime = Date.now() - startTime;
    const totalCost = model.cost_per_sec * duration;

    const metrics: RenderMetrics = {
      model_used: model.name,
      model_id: model.model_id,
      replicate_model_id: model.replicate_model_id,
      cost_per_sec: model.cost_per_sec,
      prompt_used: optimizedPrompt,
      render_time: renderTime,
      success_rate: 1.0,
      quality_score: model.performanceScore,
      total_cost: totalCost,
      api_payload: apiPayload,
      scene_assignment_reason: assignment.assignment_reason,
    };

    console.log(`     Scene ${sceneNumber} completed in ${(renderTime / 1000).toFixed(1)}
s`);
    return { videoUrl, metrics };
  }

  private buildAPIPayload(model: VideoModel, prompt: string, scene: any): Record<string,
any> {
    const basePayload = {
      prompt: prompt,
      duration: Math.min(scene.duration || 5, model.maxDuration),
      aspect_ratio: "9:16",
    };

    // Model-specific parameters
    if (model.replicate_model_id.includes("kling")) {
      return {
        ...basePayload,
        cfg_scale: model.apiParams.cfg_scale,
```

```
      negative_prompt: model.apiParams.negative_prompt,
    };
  } else if (model.replicate_model_id.includes("luma")) {
    return {
      ...basePayload,
      loop: model.apiParams.loop,
      concepts: model.apiParams.concepts,
    };
  } else if (model.replicate_model_id.includes("haiper")) {
    return {
      ...basePayload,
      use_prompt_enhancer: model.apiParams.use_prompt_enhancer,
      negative_prompt: model.apiParams.negative_prompt,
    };
  }

  return basePayload;
 }

 private async executeReplicateRender(apiPayload: Record<string, any>, model:
VideoModel): Promise<string> {
  try {
    const response = await fetch("https://api.replicate.com/v1/predictions", {
      method: "POST",
      headers: {
        Authorization: `Token ${process.env.REPLICATE_API_TOKEN}`,
        "Content-Type": "application/json",
      },
      body: JSON.stringify({
        version: model.replicate_model_id,
        input: apiPayload,
      }),
    });

    if (!response.ok) {
      throw new Error(`Replicate API error: ${response.statusText}`);
    }

    const prediction = await response.json();
    return await this.pollForCompletion(prediction.id);
  } catch (error) {
```

```
    console.error("Replicate render failed:", error);
    throw error;
  }
}

private async pollForCompletion(predictionId: string): Promise<string> {
  const maxAttempts = 180; // 15 minutes max
  let attempts = 0;

  while (attempts < maxAttempts) {
    try {
      const response = await fetch(`https://api.replicate.com/v1/predictions/${predictionId}`, {
        headers: {
          Authorization: `Token ${process.env.REPLICATE_API_TOKEN}`,
        },
      });

      const prediction = await response.json();

      if (prediction.status === "succeeded") {
        return prediction.output;
      } else if (prediction.status === "failed") {
        throw new Error(`Video generation failed: ${prediction.error}`);
      } else if (prediction.status === "canceled") {
        throw new Error("Video generation was canceled");
      }

      await new Promise((resolve) => setTimeout(resolve, 5000));
      attempts++;
    } catch (error) {
      console.error("Polling error:", error);
      attempts++;
      await new Promise((resolve) => setTimeout(resolve, 5000));
    }
  }

  throw new Error("Video generation timed out");
}

private enhancePrompt(scene: any, model: VideoModel): string {
  let prompt = scene.prompt_text || scene.narration_text;
```

```
    // Model-specific optimization
    if (model.name === "Kling Pro") {
      prompt += ". Ultra-high quality cinematic 9:16 vertical video, professional lighting, smooth motion, exceptional detail.";
    } else if (model.name === "LumaRay Flash") {
      prompt += ". Cinematic 9:16 vertical video, natural motion, physics-accurate, smooth camera movement.";
    } else if (model.name === "Haiper 2.0") {
      prompt += ". High-quality 9:16 vertical video, enhanced temporal coherence, detailed composition.";
    }

    prompt += " TikTok optimized, vertical format, high engagement potential, viral-ready content.";
    return prompt;
  }

  private async generateFallbackVideo(scene: any, sceneNumber: number): Promise<
{ videoUrl: string; metrics: RenderMetrics }> {
    // Return placeholder video for failed renders
    return {
      videoUrl: `/api/videos/fallback-scene-${sceneNumber}.mp4`,
      metrics: {
        model_used: "fallback",
        model_id: "fallback",
        replicate_model_id: "fallback",
        cost_per_sec: 0,
        prompt_used: "fallback",
        render_time: 0,
        success_rate: 0,
        quality_score: 0,
        total_cost: 0,
        api_payload: {},
        scene_assignment_reason: "Fallback for failed render",
      },
    };
  }
}
```

STEP 5: API ROUTES

5.1 Pipeline Execution Route (app/api/pipeline/execute/route.ts)
typescriptimport { NextRequest, NextResponse } from "next/server";
import { TrendHunterAgent } from "@/lib/agents/trend-hunter";
import { ScriptWriterAgent } from "@/lib/agents/script-writer";
import { ScenePlannerAgent } from "@/lib/agents/scene-planner";
import { VideoGeneratorAgent } from "@/lib/agents/video-generator";
import { PrismaClient } from "@prisma/client";

```
const prisma = new PrismaClient();

export async function POST(request: NextRequest) {
  try {
    const { nicheTags, tone } = await request.json();
    const sessionId = `session_${Date.now()}`;

    // Create pipeline execution record
    const execution = await prisma.pipelineExecution.create({
      data: {
        sessionId,
        status: "running",
        currentAgent: "trend-hunter",
        progress: 0,
        executionLogs: [],
      },
    });

    console.log(`         PIPELINE: Starting execution ${sessionId}`);

    try {
      // Step 1: Trend Discovery
      await updateProgress(sessionId, "trend-hunter", 10, "Discovering trends...");
      const trendHunter = new TrendHunterAgent();
      const trends = await trendHunter.discoverTrends(nicheTags);

      if (!trends || trends.length === 0) {
        throw new Error("No trends discovered");
      }

      // Save trends to database
      const trendPackage = await prisma.trendPackage.create({
        data: {
```

```
    trendId: trends[0].trend_id,
    trendTitle: trends[0].trend_title,
    source: trends[0].source,
    topicTags: trends[0].topic_tags,
    viralityScore: trends[0].virality_score,
    crossPlatformMatch: trends[0].cross_platform_match,
    visualStoryOpportunity: trends[0].visual_story_opportunity,
    nicheAlignment: trends[0].niche_alignment,
    monetizationPotential: trends[0].monetization_potential,
  },
});

// Step 2: Script Generation
await updateProgress(sessionId, "script-writer", 30, "Generating script...");
const scriptWriter = new ScriptWriterAgent();
const script = await scriptWriter.generateScript(trends[0], tone);

const scriptRecord = await prisma.script.create({
  data: {
    scriptId: script.script_id,
    trendPackageId: trendPackage.id,
    hookText: script.hook.text,
    hookDuration: script.hook.duration_seconds,
    bodyText: script.body.text,
    bodyDuration: script.body.duration_seconds,
    ctaText: script.cta.text,
    ctaDuration: script.cta.duration_seconds,
    totalDuration: script.metadata.total_duration,
    tone: script.metadata.tone,
    pacing: script.metadata.pacing,
    complexity: script.body.complexity,
  },
});

// Step 3: Scene Planning
await updateProgress(sessionId, "scene-planner", 50, "Planning scenes...");
const scenePlanner = new ScenePlannerAgent();
const scenePlan = await scenePlanner.planScenes(script);

const scenePlanRecord = await prisma.scenePlan.create({
  data: {
```

```
    scenePlanId: `plan_${Date.now()}`,
    scriptId: scriptRecord.id,
    segments: scenePlan.scene_plan,
    totalDuration: scenePlan.total_duration,
    visualCoherenceScore: scenePlan.visual_coherence_score,
  },
});

// Step 4: Video Generation
await updateProgress(sessionId, "video-generator", 70, "Generating videos...");
const videoGenerator = new VideoGeneratorAgent();
const videoClips = await videoGenerator.generateVideos(scenePlan);

const videoGeneration = await prisma.videoGeneration.create({
  data: {
    scenePlanId: scenePlanRecord.id,
    videoClips,
    modelDistribution: {},
    totalCost: 0, // Calculate actual cost
    avgRenderTime: 0, // Calculate actual time
    successRate: videoClips.length / scenePlan.scene_plan.length,
  },
});

// Step 5: Video Stitching (placeholder)
await updateProgress(sessionId, "stitcher", 90, "Stitching final video...");
const finalVideoUrl = await this.mockStitchVideos(videoClips);

const finalVideo = await prisma.finalVideo.create({
  data: {
    videoGenerationId: videoGeneration.id,
    finalVideoUrl,
    duration: scenePlan.total_duration,
    fileSize: 0, // Calculate actual size
    processingTime: 0, // Calculate actual time
    qualityScore: 8.5,
    captionAccuracy: 0.9,
    audioSyncScore: 0.88,
    status: "completed",
  },
});
```

```
    // Complete pipeline
    await updateProgress(sessionId, "completed", 100, "Pipeline completed successfully");

    return NextResponse.json({
      success: true,
      sessionId,
      finalVideoUrl,
      pipeline_outputs: {
        trends,
        script,
        scenePlan,
        videoClips,
        finalVideoUrl,
      },
    });
  } catch (error) {
    console.error("Pipeline execution failed:", error);

    await prisma.pipelineExecution.update({
      where: { sessionId },
      data: {
        status: "failed",
        errorMessage: error.message,
        endTime: new Date(),
      },
    });

    return NextResponse.json({
      success: false,
      error: error.message,
      sessionId,
    }, { status: 500 });
  }
} catch (error) {
  console.error("Pipeline initialization failed:", error);
  return NextResponse.json({
    success: false,
    error: "Failed to initialize pipeline",
  }, { status: 500 });
}
```

```
}

async function updateProgress(sessionId: string, agent: string, progress: number, message:
string) {
  await prisma.pipelineExecution.update({
    where: { sessionId },
    data: {
      currentAgent: agent,
      progress,
      executionLogs: {
        push: {
          timestamp: new Date(),
          agent,
          progress,
          message,
        },
      },
    },
  });
}

// Mock stitching function - replace with actual implementation
async function mockStitchVideos(videoClips: string[]): Promise<string> {
  // Placeholder for actual video stitching logic
  return `/api/videos/final_${Date.now()}.mp4`;
}

export async function GET(request: NextRequest) {
  const { searchParams } = new URL(request.url);
  const sessionId = searchParams.get("sessionId");

  if (!sessionId) {
    return NextResponse.json({ error: "Session ID required" }, { status: 400 });
  }

  try {
    const execution = await prisma.pipelineExecution.findUnique({
      where: { sessionId },
    });

    if (!execution) {
```

```
      return NextResponse.json({ error: "Session not found" }, { status: 404 });
    }

    return NextResponse.json(execution);
  } catch (error) {
    return NextResponse.json({ error: "Failed to fetch session" }, { status: 500 });
  }
}
```

5.2 Individual Agent Routes

Create individual routes for each agent in app/api/agents/[agent]/route.ts:

```
typescript// app/api/agents/trends/route.ts
import { NextRequest, NextResponse } from "next/server";
import { TrendHunterAgent } from "@/lib/agents/trend-hunter";

export async function POST(request: NextRequest) {
  try {
    const { nicheTags } = await request.json();
    const trendHunter = new TrendHunterAgent();
    const trends = await trendHunter.discoverTrends(nicheTags);

    return NextResponse.json({ success: true, trends });
  } catch (error) {
    return NextResponse.json({ success: false, error: error.message }, { status: 500 });
  }
}

// Similar patterns for other agents...
```

STEP 6: DASHBOARD COMPONENTS

6.1 Main Dashboard (app/dashboard/page.tsx)

```
typescript'use client';

import { useState, useEffect } from 'react';
import { AgentStatus } from '@/components/AgentStatus';
import { VideoPreview } from '@/components/VideoPreview';

interface PipelineExecution {
  sessionId: string;
  status: string;
  currentAgent?: string;
  progress: number;
```

```
  startTime: string;
  endTime?: string;
  errorMessage?: string;
}

export default function Dashboard() {
  const [execution, setExecution] = useState<PipelineExecution | null>(null);
  const [isRunning, setIsRunning] = useState(false);
  const [finalVideo, setFinalVideo] = useState<string | null>(null);

  const startPipeline = async () => {
    setIsRunning(true);
    setExecution(null);
    setFinalVideo(null);

    try {
      const response = await fetch('/api/pipeline/execute', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
        },
        body: JSON.stringify({
          nicheTags: ['technology', 'AI', 'future'],
          tone: 'engaging',
        }),
      });

      const result = await response.json();

      if (result.success) {
        setFinalVideo(result.finalVideoUrl);

        // Poll for progress updates
        const pollProgress = setInterval(async () => {
          const progressResponse = await fetch(`/api/pipeline/execute?sessionId=
${result.sessionId}`);
          const progressData = await progressResponse.json();

          setExecution(progressData);

          if (progressData.status === 'completed' || progressData.status === 'failed') {
```

```
          clearInterval(pollProgress);
          setIsRunning(false);
        }
      }, 2000);
    } else {
      console.error('Pipeline failed:', result.error);
      setIsRunning(false);
    }
  } catch (error) {
    console.error('Failed to start pipeline:', error);
    setIsRunning(false);
  }
};

return (
  <div className="min-h-screen bg-gray-50 p-6">
    <div className="max-w-7xl mx-auto">
      <div className="mb-8">
        <h1 className="text-3xl font-bold text-gray-900">AEON AI Video System</h1>
        <p className="text-gray-600">Autonomous AI Video Generation Pipeline</p>
      </div>

      <div className="grid grid-cols-1 lg:grid-cols-2 gap-6 mb-8">
        <div className="bg-white rounded-lg p-6 shadow-sm border border-gray-200">
          <h2 className="text-xl font-semibold mb-4">Pipeline Control</h2>

          <button
            onClick={startPipeline}
            disabled={isRunning}
            className={`w-full py-3 px-6 rounded-lg font-medium ${
              isRunning
                ? 'bg-gray-400 cursor-not-allowed'
                : 'bg-blue-600 hover:bg-blue-700 text-white'
            }`}
          >
            {isRunning ? 'Pipeline Running...' : 'Start Video Generation'}
          </button>

          {execution && (
            <div className="mt-4">
              <div className="flex justify-between items-center mb-2">
```

```
                <span className="text-sm font-medium">Progress</span>
                <span className="text-sm text-gray-500">{execution.progress}%</span>
              </div>
              <div className="w-full bg-gray-200 rounded-full h-2">
                <div
                  className="bg-blue-600 h-2 rounded-full transition-all duration-300"
                  style={{ width: `${execution.progress}%` }}
                ></div>
              </div>
              <p className="text-sm text-gray-600 mt-2">
                Current Agent: {execution.currentAgent || 'Initializing...'}
              </p>
              <p className="text-sm text-gray-600">
                Status: {execution.status}
              </p>
            </div>
          )}
        </div>

        <div className="bg-white rounded-lg p-6 shadow-sm border border-gray-200">
          <h2 className="text-xl font-semibold mb-4">System Status</h2>
          <AgentStatus />
        </div>
      </div>

      {finalVideo && (
        <div className="bg-white rounded-lg p-6 shadow-sm border border-gray-200">
          <h2 className="text-xl font-semibold mb-4">Generated Video</h2>
          <VideoPreview videoUrl={finalVideo} />
        </div>
      )}
    </div>
  </div>
 );
}
```

6.2 Agent Status Component (components/AgentStatus.tsx)

```typescript
'use client';

import { useState, useEffect } from 'react';

const agents = [
```

```
  { name: 'Trend Hunter', status: 'ready', description: 'Discovers viral content trends' },
  { name: 'Script Writer', status: 'ready', description: 'Generates optimized scripts' },
  { name: 'Scene Planner', status: 'ready', description: 'Plans visual sequences' },
  { name: 'Video Generator', status: 'ready', description: 'Creates video clips' },
  { name: 'Stitcher', status: 'ready', description: 'Combines clips into final video' },
  { name: 'Optimizer', status: 'ready', description: 'Analyzes and improves performance' },
  { name: 'Scheduler', status: 'ready', description: 'Orchestrates pipeline execution' },
];

export function AgentStatus() {
  const [agentStatuses, setAgentStatuses] = useState(agents);

  return (
    <div className="space-y-3">
      {agentStatuses.map((agent, index) => (
        <div key={index} className="flex items-center justify-between p-3 bg-gray-50
rounded-lg">
          <div>
            <h3 className="font-medium text-gray-900">{agent.name}</h3>
            <p className="text-sm text-gray-600">{agent.description}</p>
          </div>
          <div className="flex items-center">
            <div className={`w-3 h-3 rounded-full mr-2 ${
              agent.status === 'ready' ? 'bg-green-500' :
              agent.status === 'running' ? 'bg-yellow-500' :
              'bg-red-500'
            }`}></div>
            <span className="text-sm font-medium text-gray-700 capitalize">
              {agent.status}
            </span>
          </div>
        </div>
      ))}
    </div>
  );
}
```

6.3 Video Preview Component (components/VideoPreview.tsx)

```
typescript'use client';

interface VideoPreviewProps {
  videoUrl: string;
```

```
}

export function VideoPreview({ videoUrl }: VideoPreviewProps) {
  return (
    <div className="flex flex-col items-center">
      <div className="bg-black rounded-lg overflow-hidden" style={{ aspectRatio: '9/16',
width: '300px' }}>
        <video
          src={videoUrl}
          controls
          className="w-full h-full object-cover"
          poster="/video-placeholder.jpg"
        >
          Your browser does not support the video tag.
        </video>
      </div>

      <div className="mt-4 flex space-x-4">

          href={videoUrl}
          download
          className="px-4 py-2 bg-blue-600 text-white rounded-lg hover:bg-blue-700
transition-colors"
        >
          Download Video
        </a>
        <button
          onClick={() => navigator.clipboard.writeText(videoUrl)}
          className="px-4 py-2 bg-gray-600 text-white rounded-lg hover:bg-gray-700
transition-colors"
        >
          Copy URL
        </button>
      </div>
    </div>
  );
}

STEP 7: DEPLOYMENT SETUP
7.1 Next.js Configuration (next.config.js)
javascript/** @type {import('next').NextConfig} */
```

```
const nextConfig = {
  experimental: {
    serverActions: true,
  },
  images: {
    domains: ['replicate.delivery', 'pbxt.replicate.delivery'],
  },
  async headers() {
    return [
      {
        source: '/api/:path*',
        headers: [
          { key: 'Access-Control-Allow-Origin', value: '*' },
          { key: 'Access-Control-Allow-Methods', value: 'GET, POST, PUT, DELETE,
OPTIONS' },
          { key: 'Access-Control-Allow-Headers', value: 'Content-Type, Authorization' },
        ],
      },
    ];
  },
};

module.exports = nextConfig;
```

7.2 Package.json Scripts

```
json{
  "name": "aeon-video-system",
  "version": "0.1.0",
  "private": true,
  "scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "next start",
    "lint": "next lint",
    "db:generate": "prisma generate",
    "db:push": "prisma db push",
    "db:migrate": "prisma migrate dev",
    "db:studio": "prisma studio"
  },
  "dependencies": {
    "@prisma/client": "^5.0.0",
    "@vercel/blob": "^0.15.0",
```

```
    "@vercel/kv": "^1.0.0",
    "@ai-sdk/openai": "^0.0.50",
    "ai": "^3.0.0",
    "next": "14.0.0",
    "openai": "^4.0.0",
    "prisma": "^5.0.0",
    "react": "^18.0.0",
    "react-dom": "^18.0.0",
    "replicate": "^0.25.0",
    "typescript": "^5.0.0",
    "zod": "^3.22.0"
  },
  "devDependencies": {
    "@types/node": "^20.0.0",
    "@types/react": "^18.0.0",
    "@types/react-dom": "^18.0.0",
    "autoprefixer": "^10.0.1",
    "eslint": "^8.0.0",
    "eslint-config-next": "14.0.0",
    "postcss": "^8.0.0",
    "tailwindcss": "^3.3.0"
  }
}
```

STEP 8: DEPLOYMENT INSTRUCTIONS

8.1 Vercel Deployment

Push to GitHub:

```
bashgit init
git add .
git commit -m "Initial AEON system implementation"
git branch -M main
git remote add origin https://github.com/yourusername/aeon-video-system.git
git push -u origin main
```

Deploy to Vercel:

Connect your GitHub repository to Vercel

Set environment variables in Vercel dashboard

Deploy automatically on push

Set up Vercel Postgres:

Add Vercel Postgres to your project
Copy connection string to DATABASE_URL
Run npx prisma db push to create tables

8.2 Environment Variables Setup
In Vercel dashboard, add these environment variables:
OPENAI_API_KEY=your_openai_key
REPLICATE_API_TOKEN=your_replicate_token
DATABASE_URL=your_postgres_connection_string
BLOB_READ_WRITE_TOKEN=your_vercel_blob_token
KV_URL=your_vercel_kv_url
KV_REST_API_URL=your_kv_rest_url
KV_REST_API_TOKEN=your_kv_token
KV_REST_API_READ_ONLY_TOKEN=your_kv_readonly_token
WEBHOOK_SECRET=your_webhook_secret

STEP 9: TESTING & VALIDATION
9.1 Basic Testing
typescript// test/pipeline.test.ts
```
import { TrendHunterAgent } from '@/lib/agents/trend-hunter';
import { ScriptWriterAgent } from '@/lib/agents/script-writer';

describe('AEON Pipeline', () => {
  test('Trend Hunter generates valid trends', async () => {
    const trendHunter = new TrendHunterAgent();
    const trends = await trendHunter.discoverTrends(['technology']);

    expect(trends).toBeDefined();
    expect(trends.length).toBeGreaterThan(0);
    expect(trends[0]).toHaveProperty('trend_id');
    expect(trends[0]).toHaveProperty('virality_score');
  });

  test('Script Writer generates valid scripts', async () => {
    const scriptWriter = new ScriptWriterAgent();
```

```
    const mockTrend = {
      trend_id: 'test_001',
      trend_title: 'AI automation test',
      source: 'test',
      topic_tags: ['AI', 'automation'],
      virality_score: 85,
      cross_platform_match: true,
      visual_story_opportunity: 'High potential',
      niche_alignment: 'technology',
      monetization_potential: 8,
    };

    const script = await scriptWriter.generateScript(mockTrend);

    expect(script).toBeDefined();
    expect(script.hook).toBeDefined();
    expect(script.body).toBeDefined();
    expect(script.cta).toBeDefined();
    expect(script.metadata.total_duration).toBeGreaterThan(0);
  });
});
```

9.2 Manual Testing Checklist

- [ ] Pipeline executes without errors
- [ ] Each agent produces valid output
- [ ] Database records are created correctly
- [ ] Video generation works with Replicate API
- [ ] Dashboard displays pipeline progress
- [ ] Final video is accessible and playable
- [ ] Error handling works for failed operations

STEP 10: PRODUCTION CONSIDERATIONS

10.1 Scaling Optimizations

Caching: Implement Redis caching for agent outputs
Queue System: Use Vercel KV for job queuing
Rate Limiting: Implement API rate limiting
Monitoring: Add comprehensive logging and metrics

10.2 Security Enhancements

API Authentication: Implement JWT-based auth
Input Validation: Validate all user inputs
Rate Limiting: Prevent abuse of expensive operations
Secrets Management: Use Vercel environment variables

10.3 Cost Management

Model Selection: Optimize model usage based on budget
Caching: Cache expensive AI operations
Monitoring: Track costs per pipeline execution
Limits: Implement daily spending limits

DEPLOYMENT VERIFICATION
After deployment, verify these endpoints work:

GET /api/pipeline/execute?sessionId=test - Pipeline status
POST /api/pipeline/execute - Start pipeline
POST /api/agents/trends - Test trend discovery
GET /dashboard - Dashboard access

NEXT STEPS

Phase 1: Deploy basic pipeline
Phase 2: Add advanced features and optimization.
Phase 3: Scale and productionize
Phase 4: Add monetization and analytics