README.md

__init__.py

```python
"""Multi-Agent Market Intelligence System"""

from app.agents.base_agent import BaseAgent
from app.agents.researcher_agent import ResearcherAgent
from app.agents.market_agent import MarketAgent
from app.agents.evaluator_agent import EvaluatorAgent
from app.agents.idea_generator_agent import IdeaGeneratorAgent
from app.agents.orchestrator import Orchestrator
from app.agents.schemas import (
    AgentRole,
    TaskStatus,
    ResearchResult,
    MarketAnalysis,
    GeneratedIdea,
    IdeaEvaluation,
    OrchestrationResult,
    AgentTask,
    AgentMessage,
)

__all__ = [
    "BaseAgent",
    "ResearcherAgent",
    "MarketAgent",
    "EvaluatorAgent",
    "IdeaGeneratorAgent",
    "Orchestrator",
    "AgentRole",
    "TaskStatus",
    "ResearchResult",
    "MarketAnalysis",
    "GeneratedIdea",
    "IdeaEvaluation",
    "OrchestrationResult",
    "AgentTask",
    "AgentMessage",
]
```

base_agent.py

```python
from abc import ABC, abstractmethod
from typing import Any, Dict, Optional
from datetime import datetime
import uuid

from app.llm.client import LLMWrapper
from app.llm.model_router import ModelRouter
from app.agents.schemas import AgentRole, TaskStatus, AgentTask


class BaseAgent(ABC):

    def __init__(
        self,
        role: AgentRole,
        model: Optional[str] = None,
        system_prompt: Optional[str] = None
    ):
        self.role = role
        self.model = model or ModelRouter.get_model_for_task("synthesis")
        self.llm = LLMWrapper(model=self.model)
        self.system_prompt = system_prompt or self._get_default_system_prompt()
        self.task_history: list[AgentTask] = []

    @abstractmethod
    def _get_default_system_prompt(self) -> str:
        pass

    @abstractmethod
    def execute(self, task: AgentTask) -> AgentTask:
        pass

    def create_task(
        self,
        task_type: str,
        input_data: Dict[str, Any]
    ) -> AgentTask:
        task = AgentTask(
            task_id=str(uuid.uuid4()),
            agent_role=self.role,
            task_type=task_type,
            input_data=input_data,
            status=TaskStatus.PENDING,
            created_at=datetime.now()
        )
        return task

    def _update_task_status(
        self,
        task: AgentTask,
        status: TaskStatus,
        result: Optional[Any] = None,
        error: Optional[str] = None
    ) -> AgentTask:
        task.status = status
        if result:
            task.result = result
        if error:
            task.error = error
        if status == TaskStatus.COMPLETED:
            task.completed_at = datetime.now()
        return task

    def query_llm(
        self,
        prompt: str,
        system_prompt: Optional[str] = None,
        temperature: Optional[float] = None
    ) -> str:
        sys_prompt = system_prompt or self.system_prompt
        return self.llm.query(prompt, system_prompt=sys_prompt)

    def log_task(self, task: AgentTask):
        self.task_history.append(task)

    def get_task_history(self) -> list[AgentTask]:
        return self.task_history

    def __repr__(self) -> str:
        return f"{self.__class__.__name__}(role={self.role.value})"

```

evaluator_agent.py

```python
import json
from typing import List
from datetime import datetime

from app.agents.base_agent import BaseAgent
from app.agents.schemas import (
    AgentRole, AgentTask, TaskStatus, IdeaEvaluation,
    GeneratedIdea, MarketAnalysis
)
from app.llm.model_router import ModelRouter


class EvaluatorAgent(BaseAgent):

    def __init__(self):
        super().__init__(
            role=AgentRole.EVALUATOR,
            model=ModelRouter.get_model_for_task("extraction")
        )

    def _get_default_system_prompt(self) -> str:
        return """You are a senior startup evaluator and advisor with experience in:
- Product-market fit assessment
- Market opportunity evaluation
- Feasibility analysis
- Risk assessment
- Investment decision-making

Provide honest, critical, and constructive evaluations. Be specific about strengths,
weaknesses, and actionable recommendations."""

    def execute(self, task: AgentTask) -> AgentTask:
        """Execute an evaluation task"""
        try:
            self._update_task_status(task, TaskStatus.IN_PROGRESS)

            idea = task.input_data.get("idea")
            idea_obj = task.input_data.get("idea_obj")  # GeneratedIdea object
            market_analysis = task.input_data.get("market_analysis")  # MarketAnalysis object

            # Extract idea details
            if idea_obj and isinstance(idea_obj, GeneratedIdea):
                idea_text = f"""
Title: {idea_obj.title}
Description: {idea_obj.description}
Target Audience: {idea_obj.target_audience}
Value Proposition: {idea_obj.value_proposition}
Key Features: {', '.join(idea_obj.key_features)}
Market Opportunity: {idea_obj.market_opportunity}
"""
            else:
                idea_text = idea or "No idea provided"

            market_context = ""
            if market_analysis and isinstance(market_analysis, MarketAnalysis):
                market_context = f"""
Market Context:
- Market Size: {market_analysis.market_size}
- Market Maturity: {market_analysis.market_maturity}
- Trends: {', '.join(market_analysis.trends)}
- Opportunities: {', '.join(market_analysis.opportunities)}
- Threats: {', '.join(market_analysis.threats)}
- Competitive Landscape: {market_analysis.competitive_landscape}
"""

            evaluation_prompt = f"""Evaluate the following startup idea:

{idea_text}

{market_context if market_context else ""}

Provide a comprehensive evaluation in JSON format:
{{
    "feasibility_score": 0.0-1.0,
    "market_potential_score": 0.0-1.0,
    "innovation_score": 0.0-1.0,
    "overall_score": 0.0-1.0,
    "strengths": ["strength 1", "strength 2"],
    "weaknesses": ["weakness 1", "weakness 2"],
    "risks": ["risk 1", "risk 2"],
    "recommendations": ["recommendation 1", "recommendation 2"],
    "verdict": "high_potential|medium_potential|low_potential"
}}

Scoring guidelines:
- Feasibility: Can this be built? (technical, resource, time constraints)
- Market Potential: Is there a real market need? (size, growth, willingness to pay)
- Innovation: How novel/unique is this? (differentiation, competitive advantage)
- Overall: Weighted average considering all factors

Be honest and specific."""

            response = self.query_llm(evaluation_prompt)

            try:
                parsed = json.loads(response)
            except json.JSONDecodeError:
                parsed = self._parse_evaluation_fallback(response)

            overall_score = parsed.get("overall_score")
            if overall_score is None:
                overall_score = (
                    parsed.get("feasibility_score", 0.5) * 0.3 +
                    parsed.get("market_potential_score", 0.5) * 0.4 +
                    parsed.get("innovation_score", 0.5) * 0.3
                )

            result = IdeaEvaluation(
                idea=idea_text,
                feasibility_score=float(parsed.get("feasibility_score", 0.5)),
                market_potential_score=float(parsed.get("market_potential_score", 0.5)),
                innovation_score=float(parsed.get("innovation_score", 0.5)),
                overall_score=float(overall_score),
                strengths=parsed.get("strengths", []),
                weaknesses=parsed.get("weaknesses", []),
                risks=parsed.get("risks", []),
                recommendations=parsed.get("recommendations", []),
                verdict=parsed.get("verdict", "medium_potential")
            )

            self._update_task_status(task, TaskStatus.COMPLETED, result=result)
            self.log_task(task)

        except Exception as e:
            self._update_task_status(
                task,
                TaskStatus.FAILED,
                error=str(e)
            )
            self.log_task(task)

        return task

    def _parse_evaluation_fallback(self, text: str) -> dict:
        """Fallback parser if JSON parsing fails"""
        return {
            "feasibility_score": 0.5,
            "market_potential_score": 0.5,
            "innovation_score": 0.5,
            "overall_score": 0.5,
            "strengths": [],
            "weaknesses": [],
            "risks": [],
            "recommendations": [],
            "verdict": "medium_potential"
        }

    def evaluate_idea(
        self,
        idea: str = None,
        idea_obj: GeneratedIdea = None,
        market_analysis: MarketAnalysis = None
    ) -> IdeaEvaluation:
        """Convenience method for quick idea evaluation"""
        task = self.create_task(
            "idea_evaluation",
            {
                "idea": idea,
                "idea_obj": idea_obj,
                "market_analysis": market_analysis
            }
        )
        task = self.execute(task)
        return task.result

```

idea_generator_agent.py

```python
import json
from typing import List
from datetime import datetime

from app.agents.base_agent import BaseAgent
from app.agents.schemas import (
    AgentRole, AgentTask, TaskStatus, GeneratedIdea,
    ResearchResult, MarketAnalysis
)
from app.llm.model_router import ModelRouter


class IdeaGeneratorAgent(BaseAgent):
    """Agent responsible for generating startup ideas based on research"""

    def __init__(self):
        super().__init__(
            role=AgentRole.IDEA_GENERATOR,
            model=ModelRouter.get_model_for_task("synthesis")
        )

    def _get_default_system_prompt(self) -> str:
        return """You are a creative startup ideation expert with deep knowledge of:
- Emerging technologies and trends
- Market gaps and opportunities
- User pain points and unmet needs
- Successful startup patterns

Generate innovative, feasible, and market-driven startup ideas. Focus on:
- Solving real problems
- Clear value propositions
- Specific target audiences
- Differentiated features"""

    def execute(self, task: AgentTask) -> AgentTask:
        """Execute an idea generation task"""
        try:
            self._update_task_status(task, TaskStatus.IN_PROGRESS)

            topic = task.input_data.get("topic", "")
            num_ideas = task.input_data.get("num_ideas", 3)
            research_result = task.input_data.get("research_result")
            market_analysis = task.input_data.get("market_analysis")

            # Build context from research and market analysis
            context_parts = []

            if research_result and isinstance(research_result, ResearchResult):
                context_parts.append(f"""
Research Findings:
{research_result.summary}

Key Insights:
{chr(10).join(f"- {finding}" for finding in research_result.key_findings)}
""")

            if market_analysis and isinstance(market_analysis, MarketAnalysis):
                context_parts.append(f"""
Market Analysis:
- Market Size: {market_analysis.market_size}
- Maturity: {market_analysis.market_maturity}
- Trends: {', '.join(market_analysis.trends)}
- Opportunities: {', '.join(market_analysis.opportunities)}
- Threats: {', '.join(market_analysis.threats)}
""")

            context = "\n".join(context_parts) if context_parts else ""

            # Generate ideas
            generation_prompt = f"""Generate {num_ideas} innovative startup ideas related to: {topic}

{context if context else "Base ideas on general market knowledge and trends."}

For each idea, provide JSON format:
{{
    "title": "Short, catchy title",
    "description": "2-3 sentence description",
    "target_audience": "Specific target audience",
    "value_proposition": "Clear value proposition",
    "key_features": ["feature 1", "feature 2", "feature 3"],
    "market_opportunity": "Why this is a good opportunity",
    "inspiration_sources": ["source 1", "source 2"]
}}

Return as a JSON array of ideas. Make each idea:
- Specific and actionable
- Based on real pain points or opportunities
- Technically feasible
- Differentiated from existing solutions"""

            response = self.query_llm(generation_prompt, temperature=0.8)

            # Parse response
            try:
                parsed = json.loads(response)
                # Handle both array and single object responses
                if isinstance(parsed, dict):
                    parsed = [parsed]
            except json.JSONDecodeError:
                # Try to extract ideas from text
                parsed = self._parse_ideas_fallback(response)

            # Create GeneratedIdea objects
            ideas = []
            for idea_data in parsed[:num_ideas]:
                idea = GeneratedIdea(
                    title=idea_data.get("title", "Untitled Idea"),
                    description=idea_data.get("description", ""),
                    target_audience=idea_data.get("target_audience", ""),
                    value_proposition=idea_data.get("value_proposition", ""),
                    key_features=idea_data.get("key_features", []),
                    market_opportunity=idea_data.get("market_opportunity", ""),
                    inspiration_sources=idea_data.get("inspiration_sources", [])
                )
                ideas.append(idea)

            self._update_task_status(task, TaskStatus.COMPLETED, result=ideas)
            self.log_task(task)

        except Exception as e:
            self._update_task_status(
                task,
                TaskStatus.FAILED,
                error=str(e)
            )
            self.log_task(task)

        return task

    def _parse_ideas_fallback(self, text: str) -> List[dict]:
        """Fallback parser if JSON parsing fails"""
        # Simple extraction - look for numbered or bulleted ideas
        ideas = []
        lines = text.split("\n")
        current_idea = {}

        for line in lines:
            line = line.strip()
            if not line:
                continue

            if line.startswith(("1.", "2.", "3.", "4.", "5.")) or \
               (line[0].isupper() and len(line) < 100 and ":" not in line):
                if current_idea:
                    ideas.append(current_idea)
                current_idea = {"title": line.lstrip("1234567890. ")}
            elif ":" in line and current_idea:
                key, value = line.split(":", 1)
                key = key.lower().replace(" ", "_")
                current_idea[key] = value.strip()

        if current_idea:
            ideas.append(current_idea)

        return ideas if ideas else [{"title": "Generated Idea", "description": text[:200]}]

    def generate_ideas(
        self,
        topic: str,
        num_ideas: int = 3,
        research_result=None,
        market_analysis=None
    ) -> List[GeneratedIdea]:
        """Convenience method for quick idea generation"""
        task = self.create_task(
            "idea_generation",
            {
                "topic": topic,
                "num_ideas": num_ideas,
                "research_result": research_result,
                "market_analysis": market_analysis
            }
        )
        task = self.execute(task)
        return task.result if task.result else []

```

market_agent.py

```python
import json
from typing import List, Optional
from datetime import datetime

from app.agents.base_agent import BaseAgent
from app.agents.schemas import AgentRole, AgentTask, TaskStatus, MarketAnalysis, ResearchResult
from app.llm.model_router import ModelRouter


class MarketAgent(BaseAgent):

    def __init__(self):
        super().__init__(
            role=AgentRole.MARKET_ANALYST,
            model=ModelRouter.get_model_for_task("extraction")
        )

    def _get_default_system_prompt(self) -> str:
        return """You are a senior market analyst with expertise in startup ecosystems,
technology markets, and emerging trends. Your role is to:
- Analyze market dynamics and trends
- Assess market size and growth potential
- Identify opportunities and threats
- Evaluate competitive landscapes
- Determine market maturity stages
Provide data-driven, objective analysis."""

    def execute(self, task: AgentTask) -> AgentTask:
        """Execute a market analysis task"""
        try:
            self._update_task_status(task, TaskStatus.IN_PROGRESS)

            topic = task.input_data.get("topic", "")
            research_result = task.input_data.get("research_result")

            context = ""
            if research_result and isinstance(research_result, ResearchResult):
                context = f"""
Research Summary:
{research_result.summary}

Key Findings:
{chr(10).join(f"- {finding}" for finding in research_result.key_findings)}

Sources: {', '.join(research_result.sources)}
"""

            analysis_prompt = f"""Analyze the market for: {topic}

{context if context else "No prior research provided. Base your analysis on general market knowledge."}

Provide a comprehensive market analysis in JSON format:
{{
    "market_size": "Estimated market size (e.g., '$X billion', 'growing rapidly', 'unknown')",
    "trends": ["trend 1", "trend 2", "trend 3"],
    "opportunities": ["opportunity 1", "opportunity 2"],
    "threats": ["threat 1", "threat 2"],
    "competitive_landscape": "Description of competition and market structure",
    "market_maturity": "emerging|growing|mature|declining",
    "confidence_score": 0.0-1.0
}}

Be specific and data-driven where possible."""

            response = self.query_llm(analysis_prompt)

            try:
                parsed = json.loads(response)
            except json.JSONDecodeError:
                parsed = self._parse_market_analysis_fallback(response)

            result = MarketAnalysis(
                topic=topic,
                market_size=parsed.get("market_size"),
                trends=parsed.get("trends", []),
                opportunities=parsed.get("opportunities", []),
                threats=parsed.get("threats", []),
                competitive_landscape=parsed.get("competitive_landscape", ""),
                market_maturity=parsed.get("market_maturity", "unknown"),
                confidence_score=float(parsed.get("confidence_score", 0.5))
            )

            self._update_task_status(task, TaskStatus.COMPLETED, result=result)
            self.log_task(task)

        except Exception as e:
            self._update_task_status(
                task,
                TaskStatus.FAILED,
                error=str(e)
            )
            self.log_task(task)

        return task

    def _parse_market_analysis_fallback(self, text: str) -> dict:
        """Fallback parser if JSON parsing fails"""
        return {
            "market_size": "Unknown",
            "trends": [],
            "opportunities": [],
            "threats": [],
            "competitive_landscape": text[:500],
            "market_maturity": "unknown",
            "confidence_score": 0.5
        }

    def analyze_market(
        self,
        topic: str,
        research_result: Optional[ResearchResult] = None
    ) -> MarketAnalysis:
        """Convenience method for quick market analysis"""
        task = self.create_task(
            "market_analysis",
            {
                "topic": topic,
                "research_result": research_result
            }
        )
        task = self.execute(task)
        return task.result

```

orchestrator.py

```python
import time
from typing import List, Optional, Dict, Any
from datetime import datetime

from app.agents.base_agent import BaseAgent
from app.agents.researcher_agent import ResearcherAgent
from app.agents.market_agent import MarketAgent
from app.agents.evaluator_agent import EvaluatorAgent
from app.agents.idea_generator_agent import IdeaGeneratorAgent
from app.agents.schemas import (
    AgentRole, AgentTask, TaskStatus, OrchestrationResult,
    ResearchResult, MarketAnalysis, GeneratedIdea, IdeaEvaluation
)
from app.vector_db.config import VectorDBConfig


class Orchestrator:
    """Orchestrates the multi-agent system to complete complex tasks"""

    def __init__(self, vector_db_config: VectorDBConfig):
        self.researcher = ResearcherAgent(vector_db_config)
        self.market_agent = MarketAgent()
        self.evaluator = EvaluatorAgent()
        self.idea_generator = IdeaGeneratorAgent()
        self.execution_history: List[OrchestrationResult] = []

    def run_full_pipeline(
        self,
        topic: str,
        generate_ideas: bool = True,
        evaluate_ideas: bool = True,
        num_ideas: int = 5,
        top_k_research: int = 10
    ) -> OrchestrationResult:
        """
        Run the complete pipeline:
        1. Research the topic
        2. Analyze the market
        3. Generate ideas (if requested)
        4. Evaluate ideas (if requested)
        5. Return top ideas

        Args:
            topic: The topic to research and generate ideas for
            generate_ideas: Whether to generate ideas
            evaluate_ideas: Whether to evaluate generated ideas
            num_ideas: Number of ideas to generate
            top_k_research: Number of research documents to retrieve

        Returns:
            OrchestrationResult with all findings
        """
        start_time = time.time()
        result = OrchestrationResult(
            task_id=f"orchestration_{datetime.now().strftime('%Y%m%d_%H%M%S')}",
            status=TaskStatus.IN_PROGRESS
        )

        try:
            print(f"■ Researching: {topic}")
            research_task = self.researcher.create_task(
                "research",
                {"query": topic, "top_k": top_k_research}
            )
            research_task = self.researcher.execute(research_task)

            if research_task.status == TaskStatus.COMPLETED:
                result.research_results = research_task.result
                print(f"■ Research completed: {len(research_task.result.documents)} documents found")
            else:
                print(f"■ Research failed: {research_task.error}")
                result.status = TaskStatus.FAILED
                return result

            print(f"■ Analyzing market: {topic}")
            market_task = self.market_agent.create_task(
                "market_analysis",
                {
                    "topic": topic,
                    "research_result": result.research_results
                }
            )
            market_task = self.market_agent.execute(market_task)

            if market_task.status == TaskStatus.COMPLETED:
                result.market_analysis = market_task.result
                print(f"■ Market analysis completed: {market_task.result.market_maturity} market")
            else:
                print(f"■ Market analysis failed: {market_task.error}")

            if generate_ideas:
                print(f"■ Generating {num_ideas} ideas...")
                idea_task = self.idea_generator.create_task(
                    "idea_generation",
                    {
                        "topic": topic,
                        "num_ideas": num_ideas,
                        "research_result": result.research_results,
                        "market_analysis": result.market_analysis
                    }
                )
                idea_task = self.idea_generator.execute(idea_task)

                if idea_task.status == TaskStatus.COMPLETED:
                    result.generated_ideas = idea_task.result
                    print(f"■ Generated {len(idea_task.result)} ideas")
                else:
                    print(f"■ Idea generation failed: {idea_task.error}")

            if evaluate_ideas and result.generated_ideas:
                print(f"■ Evaluating {len(result.generated_ideas)} ideas...")
                evaluations = []

                for idea in result.generated_ideas:
                    eval_task = self.evaluator.create_task(
                        "idea_evaluation",
                        {
                            "idea_obj": idea,
                            "market_analysis": result.market_analysis
                        }
                    )
                    eval_task = self.evaluator.execute(eval_task)

                    if eval_task.status == TaskStatus.COMPLETED:
                        evaluations.append(eval_task.result)
                    else:
                        print(f"■■ Evaluation failed for idea: {idea.title}")

                result.evaluations = evaluations
                print(f"■ Evaluated {len(evaluations)} ideas")

                if evaluations:
                    result.top_ideas = self._rank_ideas(
                        result.generated_ideas,
                        evaluations
                    )
                    print(f"■ Top {len(result.top_ideas)} ideas selected")

            result.summary = self._generate_summary(result)
            result.execution_time = time.time() - start_time
            result.status = TaskStatus.COMPLETED

            print(f"\n■ Pipeline completed in {result.execution_time:.2f}s")

        except Exception as e:
            result.status = TaskStatus.FAILED
            result.summary = f"Pipeline failed: {str(e)}"
            result.execution_time = time.time() - start_time
            print(f"■ Pipeline failed: {str(e)}")

        self.execution_history.append(result)
        return result

    def _rank_ideas(
        self,
        ideas: List[GeneratedIdea],
        evaluations: List[IdeaEvaluation]
    ) -> List[GeneratedIdea]:
        """Rank ideas based on evaluations and return top ones"""
        if len(ideas) != len(evaluations):
            return ideas[:3]

        idea_eval_pairs = list(zip(ideas, evaluations))
        idea_eval_pairs.sort(key=lambda x: x[1].overall_score, reverse=True)

        return [idea for idea, _ in idea_eval_pairs[:3]]

    def _generate_summary(self, result: OrchestrationResult) -> str:
        """Generate a summary of the orchestration result"""
        summary_parts = []

        if result.research_results:
            summary_parts.append(
                f"Research: Found {len(result.research_results.documents)} relevant documents "
                f"with {len(result.research_results.key_findings)} key findings."
            )

        if result.market_analysis:
            summary_parts.append(
                f"Market: {result.market_analysis.market_maturity} market with "
                f"{len(result.market_analysis.opportunities)} opportunities identified."
            )

        if result.generated_ideas:
            summary_parts.append(
                f"Ideas: Generated {len(result.generated_ideas)} startup ideas."
            )

        if result.evaluations:
            avg_score = sum(e.overall_score for e in result.evaluations) / len(result.evaluations)
            summary_parts.append(
                f"Evaluation: Average score {avg_score:.2f}/1.0 across {len(result.evaluations)} ideas."
            )

        if result.top_ideas:
            summary_parts.append(
                f"Top Ideas: Selected {len(result.top_ideas)} highest-potential ideas."
            )

        return " ".join(summary_parts)

    def get_execution_history(self) -> List[OrchestrationResult]:
        """Get execution history"""
        return self.execution_history

    def __repr__(self) -> str:
        return f"Orchestrator(agents=[Researcher, Market, Evaluator, IdeaGenerator])"

```

researcher_agent.py

```python
import json
from typing import List, Dict, Any
from datetime import datetime

from app.agents.base_agent import BaseAgent
from app.agents.schemas import AgentRole, AgentTask, TaskStatus, ResearchResult
from app.rag.retriever import RAGRetriever
from app.vector_db.config import VectorDBConfig


class ResearcherAgent(BaseAgent):

    def __init__(self, vector_db_config: VectorDBConfig):
        super().__init__(role=AgentRole.RESEARCHER)
        self.retriever = RAGRetriever(vector_db_config)

    def _get_default_system_prompt(self) -> str:
        return """You are a senior research analyst specializing in startup and technology markets.
Your role is to:
- Extract key insights from research documents
- Identify patterns and trends
- Summarize findings clearly and concisely
- Highlight the most important information"""

    def execute(self, task: AgentTask) -> AgentTask:
        try:
            self._update_task_status(task, TaskStatus.IN_PROGRESS)

            query = task.input_data.get("query", "")
            top_k = task.input_data.get("top_k", 10)

            documents = self.retriever.retrieve(query, top_k=top_k)

            context = self._format_documents(documents)

            summary_prompt = f"""Based on the following research documents, provide:
1. A comprehensive summary (2-3 paragraphs)
2. Key findings (bullet points)

Query: {query}

Documents:
{context}

Format your response as JSON:
{{
    "summary": "...",
    "key_findings": ["...", "..."]
}}"""

            response = self.query_llm(summary_prompt)

            try:
                parsed = json.loads(response)
                summary = parsed.get("summary", "")
                key_findings = parsed.get("key_findings", [])
            except json.JSONDecodeError:
                summary = response
                key_findings = self._extract_findings_from_text(response)

            sources = [doc.metadata.get("source", "unknown") for doc in documents]
            sources = list(set(sources))

            result = ResearchResult(
                query=query,
                documents=[
                    {
                        "id": doc.id,
                        "text": doc.text,
                        "metadata": doc.metadata,
                        "score": doc.score
                    }
                    for doc in documents
                ],
                summary=summary,
                key_findings=key_findings,
                sources=sources,
                timestamp=datetime.now()
            )

            self._update_task_status(task, TaskStatus.COMPLETED, result=result)
            self.log_task(task)

        except Exception as e:
            self._update_task_status(
                task,
                TaskStatus.FAILED,
                error=str(e)
            )
            self.log_task(task)

        return task

    def _format_documents(self, documents: List) -> str:
        """Format documents for LLM context"""
        formatted = []
        for i, doc in enumerate(documents, 1):
            formatted.append(
                f"Document {i} (Relevance: {doc.score:.3f}):\n"
                f"Source: {doc.metadata.get('source', 'unknown')}\n"
                f"Content: {doc.text[:500]}...\n"
            )
        return "\n".join(formatted)

    def _extract_findings_from_text(self, text: str) -> List[str]:
        """Extract findings from text if JSON parsing fails"""
        lines = text.split("\n")
        findings = []
        for line in lines:
            line = line.strip()
            if line and (line.startswith("-") or line.startswith("*") or
                        (line[0].isdigit() and "." in line[:3])):
                findings.append(line.lstrip("-* ").split(". ", 1)[-1])
        return findings[:10]

    def research(self, query: str, top_k: int = 10) -> ResearchResult:
        """Convenience method for quick research"""
        task = self.create_task("research", {"query": query, "top_k": top_k})
        task = self.execute(task)
        return task.result

```

schemas.py

```python
from dataclasses import dataclass
from typing import List, Dict, Any, Optional
from datetime import datetime
from enum import Enum


class AgentRole(str, Enum):
    RESEARCHER = "researcher"
    MARKET_ANALYST = "market_analyst"
    EVALUATOR = "evaluator"
    IDEA_GENERATOR = "idea_generator"


class TaskStatus(str, Enum):
    PENDING = "pending"
    IN_PROGRESS = "in_progress"
    COMPLETED = "completed"
    FAILED = "failed"


@dataclass
class ResearchResult:
    query: str
    documents: List[Dict[str, Any]]
    summary: str
    key_findings: List[str]
    sources: List[str]
    timestamp: datetime


@dataclass
class MarketAnalysis:
    topic: str
    market_size: Optional[str]
    trends: List[str]
    opportunities: List[str]
    threats: List[str]
    competitive_landscape: str
    market_maturity: str
    confidence_score: float


@dataclass
class IdeaEvaluation:
    idea: str
    feasibility_score: float
    market_potential_score: float
    innovation_score: float
    overall_score: float
    strengths: List[str]
    weaknesses: List[str]
    risks: List[str]
    recommendations: List[str]
    verdict: str


@dataclass
class GeneratedIdea:
    title: str
    description: str
    target_audience: str
    value_proposition: str
    key_features: List[str]
    market_opportunity: str
    inspiration_sources: List[str]


@dataclass
class AgentTask:
    task_id: str
    agent_role: AgentRole
    task_type: str
    input_data: Dict[str, Any]
    status: TaskStatus = TaskStatus.PENDING
    result: Optional[Any] = None
    error: Optional[str] = None
    created_at: datetime = None
    completed_at: Optional[datetime] = None


@dataclass
class AgentMessage:
    from_agent: AgentRole
    to_agent: AgentRole
    message_type: str
    content: Dict[str, Any]
    timestamp: datetime


@dataclass
class OrchestrationResult:
    task_id: str
    research_results: Optional[ResearchResult] = None
    market_analysis: Optional[MarketAnalysis] = None
    generated_ideas: List[GeneratedIdea] = None
    evaluations: List[IdeaEvaluation] = None
    top_ideas: List[GeneratedIdea] = None
    summary: str = ""
    execution_time: float = 0.0
    status: TaskStatus = TaskStatus.PENDING

```

settings.py

```python
from functools import lru_cache
from typing import Literal

from pydantic_settings import BaseSettings, SettingsConfigDict


# =========================
# Model Configuration
# =========================
class ModelConfig(BaseSettings):
    CHAT_MODEL: str = "gpt-4o-mini"
    EMBEDDING_MODEL: str = "text-embedding-3-small"
    TEMPERATURE: float = 0.3
    MAX_TOKENS: int = 1000


class ModelRoutingConfig(BaseSettings):
    IDEATION_MODEL: str = "gpt-4o-mini"
    ANALYSIS_MODEL: str = "gpt-4o"
    SUMMARIZATION_MODEL: str = "gpt-4o-mini"


# =========================
# Vector Database Config
# =========================
class VectorDBConfig(BaseSettings):
    PROVIDER: Literal["chroma", "pinecone", "qdrant"] = "chroma"
    COLLECTION_NAME: str = "documents"
    PERSIST_DIR: str = "./vector_store"


# =========================
# Logging + Observability
# =========================
class LoggingConfig(BaseSettings):
    LOG_LEVEL: Literal["DEBUG", "INFO", "WARNING", "ERROR", "CRITICAL"] = "INFO"
    LOG_FILE: str = "./logs/app.log"


class ObservabilityConfig(BaseSettings):
    ENABLE_TRACING: bool = True
    TRACE_LLM_CALLS: bool = True
    TRACE_PIPELINES: bool = True


# =========================
# Scraper Config
# =========================
class ScraperConfig(BaseSettings):
    REDDIT_LIMIT: int = 100
    HN_LIMIT: int = 100
    X_LIMIT: int = 100
    REQUEST_TIMEOUT: int = 10
    RETRIES: int = 3
    RATE_LIMIT_PER_MINUTE: int = 30
    BACKOFF_FACTOR: float = 1.5


# =========================
# LLM Control
# =========================
class LLMConfig(BaseSettings):
    MAX_RETRIES: int = 3
    TIMEOUT: int = 60
    COST_LIMIT_PER_RUN: float = 5.0


# =========================
# Pipeline Config
# =========================
class PipelineConfig(BaseSettings):
    BATCH_SIZE: int = 32
    EMBEDDING_BATCH_SIZE: int = 64
    ENABLE_PARALLEL_STAGES: bool = True


# =========================
# Runtime Flags
# =========================
class RuntimeFlags(BaseSettings):
    ENABLE_LLM: bool = True
    ENABLE_SCRAPING: bool = True
    DRY_RUN: bool = False


# =========================
# Cache Config
# =========================
class CacheConfig(BaseSettings):
    ENABLE_CACHE: bool = True
    CACHE_DIR: str = "./cache"


# =========================
# Worker / Async Config
# =========================
class WorkerConfig(BaseSettings):
    MAX_CONCURRENT_TASKS: int = 5
    QUEUE_SIZE: int = 100


# =========================
# Paths Config
# =========================
class Paths(BaseSettings):
    RAW_DATA: str = "data/raw"
    PROCESSED_DATA: str = "data/processed"
    REPORTS: str = "reports"


# =========================
# Root Application Settings
# =========================
class AppSettings(BaseSettings):
    model_config = SettingsConfigDict(env_file=".env")

    OPENAI_API_KEY: str
    PINECONE_API_KEY: str | None = None
    ENV: Literal["development", "staging", "production"] = "development"

    model: ModelConfig = ModelConfig()
    model_routing: ModelRoutingConfig = ModelRoutingConfig()
    vectordb: VectorDBConfig = VectorDBConfig()

    logging: LoggingConfig = LoggingConfig()
    observability: ObservabilityConfig = ObservabilityConfig()

    scraper: ScraperConfig = ScraperConfig()
    llm: LLMConfig = LLMConfig()
    pipeline: PipelineConfig = PipelineConfig()

    runtime_flags: RuntimeFlags = RuntimeFlags()
    cache: CacheConfig = CacheConfig()
    worker: WorkerConfig = WorkerConfig()
    paths: Paths = Paths()


# =========================
# Settings Factory (Singleton)
# =========================
@lru_cache
def get_settings() -> AppSettings:
    return AppSettings()


# default instance
settings = get_settings()
```

hn_scraper.py

```python
import requests
import json
import os
from datetime import datetime
from app.config.settings import get_settings

settings = get_settings()

HN_API_URL = "http://hn.algolia.com/api/v1/search?tags=front_page"

def scrape_hackernews(limit=100):
    all_posts = []

    response = requests.get(HN_API_URL)
    response.raise_for_status()
    data = response.json()

    for post in data.get("hits", [])[:limit]:
        all_posts.append({
            "title": post.get("title"),
            "body": post.get("story_text") or "",
            "score": post.get("points"),
            "comments": post.get("num_comments"),
            "created_at": datetime.fromtimestamp(post.get("created_at_i")).isoformat()
            if post.get("created_at_i") else None,
            "source": "hackernews"
        })

    file_path = os.path.join(settings.paths.RAW_DATA, "hn.json")
    with open(file_path, "w", encoding="utf-8") as f:
        json.dump(all_posts, f, indent=2, ensure_ascii=False)

    print(f"Scraped {len(all_posts)} Hacker News posts -> {file_path}")
    return file_path

if __name__ == "__main__":
    scrape_hackernews(limit=settings.scraper.HN_LIMIT)

```

reddit_scraper.py

```python
import json
import os
from datetime import datetime
import praw
from app.config.settings import get_settings

settings = get_settings()

SUBREDDITS = ['startups', 'Entrepreneur', 'SaaS', 'SideProject']

def scrape_reddit(limit=100):
    reddit = praw.Reddit(
        client_id=os.getenv("REDDIT_CLIENT_ID"),
        client_secret=os.getenv("REDDIT_CLIENT_SECRET"),
        user_agent=os.getenv("REDDIT_USER_AGENT")
    )

    all_posts = []

    for sub in SUBREDDITS:
        subreddit = reddit.subreddit(sub)
        for post in subreddit.hot(limit=limit):
            all_posts.append({
                "title": post.title,
                "body": post.selftext,
                "score": post.score,
                "comments": post.num_comments,
                "created_at": datetime.fromtimestamp(post.created_utc).isoformat(),
                "source": f"reddit/r/{sub}"
            })

    file_path = os.path.join(settings.paths.RAW_DATA, "reddit.json")
    os.makedirs(os.path.dirname(file_path), exist_ok=True)
    with open(file_path, "w", encoding="utf-8") as f:
        json.dump(all_posts, f, ensure_ascii=False)

    print(f"Scraped {len(all_posts)} Reddit posts -> {file_path}")
    return file_path

if __name__ == "__main__":
    scrape_reddit(limit=settings.scraper.REDDIT_LIMIT)

```

test_json.py

```python
import json
import os
from app.config.settings import get_settings

settings = get_settings()

def test_json(file_name):
    path = os.path.join(settings.paths.RAW_DATA, file_name)
    try:
        with open(path, "r", encoding="utf-8") as f:
            data = json.load(f)
        print(f"{file_name} is valid JSON with {len(data)} records")
    except Exception as e:
        print(f"{file_name} is invalid JSON! Error: {e}")

if __name__ == "__main__":
    test_json("reddit.json")
    test_json("hn.json")

```

cache_manager.py

```python
import hashlib
import json
from app.config.settings import get_settings

settings = get_settings()

class CacheManager:
    def __init__(self):
        self.cache = {}

    def _generate_key(self, model: str, messages: list) -> str:
        data = f"{model}:{json.dumps(messages, sort_keys=True)}"
        return hashlib.md5(data.encode()).hexdigest()

    def get(self, model: str, messages: list):
        key = self._generate_key(model, messages)
        return self.cache.get(key)

    def set(self, model: str, messages: list, response: str):
        key = self._generate_key(model, messages)
        if settings.cache.ENABLE_CACHE:
            self.cache[key] = response

```

client.py

```python
from app.config.settings import get_settings
from app.llm.cache_manager import CacheManager
from app.llm.retry_handler import retry_with_backoff
from app.llm.model_router import ModelRouter
from openai import OpenAI

settings = get_settings()

class LLMWrapper:
    def __init__(self, model: str = None):
        self.model = model or settings.model_routing.IDEATION_MODEL
        self.cache = CacheManager()
        self.client = OpenAI(api_key=settings.OPENAI_API_KEY)

    @retry_with_backoff(retries=settings.llm.MAX_RETRIES, backoff_in_seconds=1)
    def query(self, prompt: str, system_prompt="You are a helpful assistant"):
        cached = self.cache.get(self.model, [{"role": "user", "content": prompt}])
        if cached:
            return cached

        response = self.client.chat.completions.create(
            model=self.model,
            messages=[
                {"role": "system", "content": system_prompt},
                {"role": "user", "content": prompt}
            ],
            temperature=settings.model.TEMPERATURE,
            max_tokens=settings.model.MAX_TOKENS
        )

        answer = response.choices[0].message.content
        self.cache.set(self.model, [{"role": "user", "content": prompt}], answer)
        return answer
```

model_router.py

```python
from app.config.settings import get_settings

settings = get_settings()

class ModelRouter:
    MODEL_MAP = {
        "synthesis": settings.model_routing.IDEATION_MODEL,
        "extraction": settings.model_routing.ANALYSIS_MODEL,
        "classification": settings.model_routing.CLASSIFICATION_MODEL
        if hasattr(settings.model_routing, "CLASSIFICATION_MODEL")
        else settings.model_routing.ANALYSIS_MODEL
    }

    @classmethod
    def get_model_for_task(cls, task_type: str):
        return cls.MODEL_MAP.get(task_type, settings.model_routing.IDEATION_MODEL)

```

template.py

```python
MARKET_ANALYSIS_TEMPLATE = """
Analyze the following market data for {sector}:
Data: {data}
Focus on: {key_metrics}
"""

def format_analysis_prompt(sector: str, data: str, key_metrics: str) -> str:
    return MARKET_ANALYSIS_TEMPLATE.format(
        sector=sector,
        data=data,
        key_metrics=key_metrics
    )

```

retry_handler.py

```python
import time
from functools import wraps

def retry_with_backoff(retries=3, backoff_in_seconds=1):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            attempts = 0
            while attempts < retries:
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    attempts += 1
                    print(f"Error: {e}. Retrying {attempts}/{retries}...")
                    time.sleep(backoff_in_seconds * (2 ** (attempts - 1)))
            return func(*args, **kwargs)
        return wrapper
    return decorator


```

main.py

```

```

context_compressor.py

```python
from typing import List
from app.rag.schemas import RetrievedDocument


class ContextCompressor:
    def __init__(self, max_chars: int = 8000):
        self.max_chars = max_chars

    def compress(self, docs: List[RetrievedDocument]) -> str:
        context = ""

        for doc in docs:
            chunk = f"[Source:{doc.metadata.get('source')}]\n{doc.text}\n\n"

            if len(context) + len(chunk) > self.max_chars:
                break

            context += chunk

        return context.strip()

```

insight_generator.py

```python
from app.llm.client import LLMWrapper
from app.llm.model_router import ModelRouter
from app.rag.schemas import InsightResult
import json


class InsightGenerator:
    def __init__(self):
        model = ModelRouter.get_model_for_task("synthesis")
        self.llm = LLMWrapper(model=model)

    def generate(self, query: str, context: str) -> InsightResult:

        system_prompt = """
You are a senior startup analyst.
Extract:
- key pain points
- market opportunities
- emerging trends
- strong signals
Return JSON.
"""

        prompt = f"""
QUERY:
{query}

DATA:
{context}

OUTPUT FORMAT:
{{
 "summary": "",
 "pain_points": [],
 "opportunities": [],
 "signals": []
}}
"""

        response = self.llm.query(prompt, system_prompt=system_prompt)

        data = json.loads(response)

        return InsightResult(
            summary=data.get("summary", ""),
            pain_points=data.get("pain_points", []),
            opportunities=data.get("opportunities", []),
            signals=data.get("signals", []),
        )

```

painpoint_clusterer.py

```python
from collections import defaultdict
from typing import List


class PainPointClusterer:
    def cluster(self, pain_points: List[str]):

        clusters = defaultdict(list)

        for p in pain_points:
            key = p.split()[0]
            clusters[key].append(p)

        return dict(clusters)

```

retriever.py

```python
from typing import List
from app.vector_db.semantic_search import SemanticSearchEngine
from app.vector_db.config import VectorDBConfig
from app.rag.schemas import RetrievedDocument


class RAGRetriever:
    def __init__(self, config: VectorDBConfig):
        self.engine = SemanticSearchEngine(config)

    def retrieve(self, query: str, top_k: int = 10) -> List[RetrievedDocument]:
        results = self.engine.search(query, top_k=top_k)

        docs = []

        ids = results["ids"][0]
        texts = results["documents"][0]
        metas = results["metadatas"][0]
        scores = results["distances"][0]

        for i in range(len(ids)):
            docs.append(
                RetrievedDocument(
                    id=ids[i],
                    text=texts[i],
                    metadata=metas[i],
                    score=scores[i],
                )
            )

        return docs

```

schemas.py

```python
from dataclasses import dataclass
from typing import List, Dict, Any


@dataclass
class RetrievedDocument:
    id: str
    text: str
    metadata: Dict[str, Any]
    score: float


@dataclass
class InsightResult:
    summary: str
    pain_points: List[str]
    opportunities: List[str]
    signals: List[str]

```

trend_detector.py

```python
from collections import Counter
from typing import List
from app.rag.schemas import RetrievedDocument


class TrendDetector:
    def detect(self, docs: List[RetrievedDocument], top_k=5):

        words = []

        for d in docs:
            words.extend(d.text.split())

        counter = Counter(words)

        return counter.most_common(top_k)

```

cache.py

```python
import json
from pathlib import Path


class EmbeddingCache:
    def __init__(self, path="data/cache/embeddings.json"):
        self.path = Path(path)
        self.path.parent.mkdir(parents=True, exist_ok=True)

        if self.path.exists():
            self.cache = json.loads(self.path.read_text())
        else:
            self.cache = {}

    def get(self, key):
        return self.cache.get(key)

    def set(self, key, value):
        self.cache[key] = value
        self.path.write_text(json.dumps(self.cache))

```

chroma_client.py

```python
import chromadb
from chromadb.config import Settings
from app.vector_db.config import VectorDBConfig


class ChromaVectorDB:
    def __init__(self, config: VectorDBConfig):
        self.config = config
        self.client = chromadb.Client(Settings(
            chroma_db_impl="duckdb+parquet",
            persist_directory=self.config.persist_directory,
            anonymized_telemetry=False
        ))
        self.collection = self.client.get_or_create_collection(name=self.config.collection_name)

    def add_documents(self, ids: list[str], documents: list[str], embeddings: list[list[float]], metadatas: list[dict] | None = N
        self.collection.add(
            ids=ids,
            documents=documents,
            embeddings=embeddings,
            metadatas=metadatas
        )

    def query(self, query_embedding, top_k=5):
        return self.collection.query(
            query_embeddings=[query_embedding],
            n_results=top_k
        )


```

config.py

```python
from dataclasses import dataclass

@dataclass
class VectorDBConfig:
    persist_directory: str = "data/vector_db"
    collection_name: str = "startup_ideas"
    embedding_model: str = "text-embedding-3-small"
    batch_size: int = 32
    max_retries: int = 3

```

deduplicator.py

```
import hashlib


def hash_text(text: str) -> str:
    return hashlib.sha256(text.encode()).hexdigest()

```

embedding_service.py

```python
import time
from openai import OpenAI
from app.vector_db.logger import get_logger
from app.config.settings import get_settings

settings = get_settings()
logger = get_logger("embedding_service")


class EmbeddingService:
    def __init__(self, model: str, max_retries=3):
        self.client = OpenAI(api_key=settings.OPENAI_API_KEY)
        self.model = model
        self.max_retries = max_retries

    def embed_batch(self, texts):
        for attempt in range(self.max_retries):
            try:
                response = self.client.embeddings.create(
                    model=self.model,
                    input=texts,
                )
                return [e.embedding for e in response.data]

            except Exception as e:
                wait = 2 ** attempt
                logger.warning(f"Embedding failed, retrying in {wait}s | {e}")
                time.sleep(wait)

        raise RuntimeError("Embedding failed after retries")

```

ingestion_pipeline.py

```python
import json
from tqdm import tqdm

from app.vector_db.schemas import Document
from app.vector_db.deduplicator import hash_text
from app.vector_db.embedding_service import EmbeddingService
from app.vector_db.cache import EmbeddingCache
from app.vector_db.chroma_client import ChromaVectorDB
from app.vector_db.logger import get_logger
from app.vector_db.config import VectorDBConfig


logger = get_logger("ingestion_pipeline")


class VectorIngestionPipeline:
    def __init__(self, config: VectorDBConfig):
        self.config = config
        self.db = ChromaVectorDB(config)
        self.embedder = EmbeddingService(
            model=config.embedding_model,
            max_retries=config.max_retries,
        )
        self.cache = EmbeddingCache()

    def load_json(self, path: str):
        with open(path, "r", encoding="utf-8") as f:
            data = json.load(f)

        documents = []

        for item in data:
            text = f"{item.get('title','')} {item.get('body','')}".strip()

            doc = Document(
                id=hash_text(text),
                text=text,
                metadata={
                    "source": item.get("source"),
                    "score": item.get("score"),
                    "comments": item.get("comments"),
                    "created_at": item.get("created_at"),
                },
            )
            documents.append(doc)

        return documents

    def index(self, documents):
        batch_size = self.config.batch_size

        for i in tqdm(range(0, len(documents), batch_size)):
            batch = documents[i : i + batch_size]

            texts = []
            ids = []
            metas = []
            embeddings = []

            for doc in batch:
                cached = self.cache.get(doc.id)

                if cached:
                    embeddings.append(cached)
                else:
                    texts.append(doc.text)

                ids.append(doc.id)
                metas.append(doc.metadata)

            if texts:
                new_embeddings = self.embedder.embed_batch(texts)

                for text, emb in zip(texts, new_embeddings):
                    self.cache.set(hash_text(text), emb)
                    embeddings.append(emb)

            self.db.add_documents(
                ids=ids,
                embeddings=embeddings,
                documents=[d.text for d in batch],
                metadatas=metas,
            )

            logger.info(f"Indexed batch size={len(batch)}")

```

logger.py

```python
import logging

def get_logger(name: str):
    logger = logging.getLogger(name)
    logger.setLevel(logging.INFO)

    if not logger.handlers:
        handler = logging.StreamHandler()
        formatter = logging.Formatter(
            '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
        )
        handler.setFormatter(formatter)
        logger.addHandler(handler)

    return logger

```

schemas.py

```python
from dataclasses import dataclass
from typing import Dict, Any

@dataclass
class Document:
    id: str
    text: str
    metadata: Dict[str, Any]

```

semantic_search.py

```python
from openai import OpenAI
from app.vector_db.chroma_client import ChromaVectorDB
from app.vector_db.config import VectorDBConfig
from app.config.settings import get_settings

settings = get_settings()


class SemanticSearchEngine:
    def __init__(self, config: VectorDBConfig):
        self.client = OpenAI(api_key=settings.OPENAI_API_KEY)
        self.model = config.embedding_model
        self.db = ChromaVectorDB(config)

    def embed_query(self, query):
        response = self.client.embeddings.create(
            model=self.model,
            input=[query],
        )
        return response.data[0].embedding

    def search(self, query, top_k=5):
        embedding = self.embed_query(query)
        return self.db.query(embedding, top_k)

```

similarity.py

```python
import math


def cosine_similarity(a, b):
    dot = sum(x * y for x, y in zip(a, b))
    na = math.sqrt(sum(x * x for x in a))
    nb = math.sqrt(sum(y * y for y in b))
    return dot / (na * nb) if na and nb else 0.0

```

generate_pdf.py

```
import os
from reportlab.platypus import SimpleDocTemplate, Paragraph, Spacer, Preformatted, PageBreak
from reportlab.lib.styles import getSampleStyleSheet, ParagraphStyle
from reportlab.lib.pagesizes import A4
from reportlab.lib.units import mm
from reportlab.lib import colors
from pygments import highlight
from pygments.lexers import get_lexer_for_filename, TextLexer
from pygments.formatters import HtmlFormatter
from pygments.styles import get_style_by_name
from reportlab.lib.styles import StyleSheet1
from reportlab.platypus import Flowable
from reportlab.lib.enums import TA_CENTER
from reportlab.lib.styles import ParagraphStyle
from reportlab.lib.styles import getSampleStyleSheet
from reportlab.lib.styles import ParagraphStyle
from reportlab.platypus import XPreformatted


EXCLUDE_DIRS = {".git", "__pycache__", "venv", ".venv", "node_modules", "build", "dist"}
SUPPORTED_EXTENSIONS = (".py", ".js", ".ts", ".go", ".java", ".cpp", ".c", ".json", ".yaml", ".yml", ".md")


def collect_code_files(directory):
    code_files = []
    for root, dirs, files in os.walk(directory):
        dirs[:] = [d for d in dirs if d not in EXCLUDE_DIRS]

        for file in files:
            if file.endswith(SUPPORTED_EXTENSIONS):
                code_files.append(os.path.join(root, file))

    return sorted(code_files)


def highlight_code(code, filename):
    try:
        lexer = get_lexer_for_filename(filename)
    except:
        lexer = TextLexer()

    formatter = HtmlFormatter(style="monokai", noclasses=True)
    return highlight(code, lexer, formatter)


def strip_html_tags(text):
    import re
    clean = re.compile("<.*?>")
    return re.sub(clean, "", text)


def create_pdf_from_code(files, output_pdf):
    doc = SimpleDocTemplate(
        output_pdf,
        pagesize=A4,
        rightMargin=20,
        leftMargin=20,
        topMargin=20,
        bottomMargin=20,
    )

    styles = getSampleStyleSheet()

    title_style = ParagraphStyle(
        name="TitleStyle",
        fontName="Helvetica-Bold",
        fontSize=12,
        spaceAfter=8,
        alignment=TA_CENTER,
    )

    code_style = ParagraphStyle(
        name="CodeStyle",
        fontName="Courier",
        fontSize=7,
        leading=8,
        backColor=None,
    )

    elements = []

    for file in files:
        with open(file, "r", encoding="utf-8", errors="ignore") as f:
            code = f.read()

        elements.append(Paragraph(os.path.basename(file), title_style))

        highlighted = highlight_code(code, file)
        clean_code = strip_html_tags(highlighted)

        numbered_lines = "\n".join(
            f"{str(i+1).rjust(4)} | {line}"
            for i, line in enumerate(clean_code.splitlines())
        )

        elements.append(XPreformatted(numbered_lines, code_style))
        elements.append(PageBreak())

    doc.build(elements)


if __name__ == "__main__":
    project_dir = "./"
    output_pdf = "project_code_pro.pdf"

    code_files = collect_code_files(project_dir)

    print(f"Found {len(code_files)} files")
    create_pdf_from_code(code_files, output_pdf)

    print(f"PDF created: {output_pdf}")

```

data_pipeline.py

```python
import json
import re
from pathlib import Path
from typing import Iterable, List

from app.config.settings import get_settings
from app.llm.client import LLMWrapper
from app.llm.model_router import ModelRouter
from app.llm.prompts.template import format_analysis_prompt

settings = get_settings()

DEFAULT_SECTOR = "startups & emerging tech"

DEFAULT_KEY_METRICS = (
    "pain points, unmet needs, traction signals, "
    "pricing hints, and competitive alternatives"
)

def load_json_records(path: Path) -> List[dict]:
    if not path.exists():
        raise FileNotFoundError(f"File not found: {path}")

    with path.open("r", encoding="utf-8") as handle:
        return json.load(handle)

def preprocess_text(title: str | None, body: str | None) -> str:
    combined = " ".join([segment for segment in [title, body] if segment]).strip()

    combined = combined.lower()
    combined = re.sub(r"http\S+|www\.\S+", "", combined)
    combined = re.sub(r"[^\w\s\-\.,!?]", "", combined)
    combined = re.sub(r"\s+", " ", combined)

    return combined.strip()

def iter_batches(items: List[str], batch_size: int) -> Iterable[List[str]]:
    for start in range(0, len(items), batch_size):
        yield items[start:start + batch_size]

def generate_embeddings(texts: List[str]) -> List[List[float]]:
    if not texts:
        return []

    from openai import OpenAI

    client = OpenAI(api_key=settings.OPENAI_API_KEY)

    embeddings: List[List[float]] = []

    for batch in iter_batches(texts, batch_size=50):
        response = client.embeddings.create(
            model=settings.model.EMBEDDING_MODEL,
            input=batch
        )

        ordered = sorted(response.data, key=lambda item: item.index)
        embeddings.extend([item.embedding for item in ordered])

    return embeddings

def generate_market_analysis(llm: LLMWrapper, text: str, source: str) -> str:
    prompt = format_analysis_prompt(
        sector=DEFAULT_SECTOR,
        data=f"Source: {source}\n\nContent: {text}",
        key_metrics=DEFAULT_KEY_METRICS,
    )

    return llm.query(
        prompt,
        system_prompt=(
            "You are a senior market research analyst. "
            "Extract startup pain points, unmet needs, and market signals."
        ),
    )

def build_processed_records(raw_records: List[dict]) -> List[dict]:

    cleaned_texts = [
        preprocess_text(record.get("title"), record.get("body"))
        for record in raw_records
    ]

    embeddings = generate_embeddings(cleaned_texts)

    analysis_model = ModelRouter.get_model_for_task("synthesis")
    llm = LLMWrapper(model=analysis_model)

    processed_records: List[dict] = []

    for record, cleaned_text, embedding in zip(
        raw_records, cleaned_texts, embeddings
    ):
        processed_records.append(
            {
                "title": record.get("title"),
                "body": record.get("body"),
                "score": record.get("score"),
                "comments": record.get("comments"),
                "created_at": record.get("created_at"),
                "source": record.get("source"),
                "cleaned_text": cleaned_text,
                "embedding": embedding,
                "market_analysis": generate_market_analysis(
                    llm,
                    cleaned_text,
                    record.get("source", "unknown"),
                ),
            }
        )

    return processed_records

def run_pipeline() -> Path:
    raw_path = Path(settings.paths.RAW_DATA)
    processed_path = Path(settings.paths.PROCESSED_DATA)
    processed_path.mkdir(parents=True, exist_ok=True)

    reddit_records = load_json_records(raw_path / "reddit.json")
    hn_records = load_json_records(raw_path / "hn.json")

    combined_records = reddit_records + hn_records

    processed_records = build_processed_records(combined_records)

    output_file = processed_path / "market_analysis.json"

    with output_file.open("w", encoding="utf-8") as handle:
        json.dump(processed_records, handle, ensure_ascii=False, indent=2)

    print(f"Processed {len(processed_records)} records -> {output_file}")

    return output_file


if __name__ == "__main__":
    run_pipeline()

```

inference_pipeline.py

```


```

training_pipeline.py

```


```

generate_report.py

```
"""
Script to generate market intelligence reports

This script can generate reports in multiple ways:
1. From processed data (market_analysis.json)
2. Using the multi-agent system for comprehensive analysis
3. Using the RAG insight generator

Usage:
    python scripts/generate_report.py [--method METHOD] [--topic TOPIC] [--output OUTPUT]

Options:
    --method: 'data' (from processed data), 'agents' (multi-agent), 'rag' (RAG insights)
    --topic: Topic for agent-based or RAG-based reports
    --output: Output file path (default: reports/report_YYYYMMDD_HHMMSS.json)
"""

import os
import sys
import json
import argparse
from pathlib import Path
from datetime import datetime

# Add project root to path
project_root = Path(__file__).parent.parent
sys.path.insert(0, str(project_root))

from app.config.settings import get_settings
from app.rag.insight_generator import InsightGenerator
from app.rag.retriever import RAGRetriever
from app.vector_db.config import VectorDBConfig
from app.agents import Orchestrator

settings = get_settings()


def generate_report_from_data(output_path: Path) -> dict:
    """Generate report from processed data"""
    processed_file = Path(settings.paths.PROCESSED_DATA) / "market_analysis.json"

    if not processed_file.exists():
        raise FileNotFoundError(f"Processed data not found: {processed_file}")

    with open(processed_file, "r", encoding="utf-8") as f:
        data = json.load(f)

    # Aggregate insights
    all_analyses = [record.get("market_analysis", "") for record in data]

    # Count sources
    sources = {}
    for record in data:
        source = record.get("source", "unknown")
        sources[source] = sources.get(source, 0) + 1

    # Generate summary using LLM
    from app.llm.client import LLMWrapper
    from app.llm.model_router import ModelRouter

    model = ModelRouter.get_model_for_task("synthesis")
    llm = LLMWrapper(model=model)

    summary_prompt = f"""Based on {len(data)} market analysis records, generate a comprehensive market intelligence report.

The report should include:
1. Executive Summary
2. Key Market Trends
3. Pain Points Identified
4. Opportunities
5. Recommendations

Analysis excerpts:
{chr(10).join(all_analyses[:10])}

Generate a structured report in JSON format:
{{
    "executive_summary": "...",
    "key_trends": ["...", "..."],
    "pain_points": ["...", "..."],
    "opportunities": ["...", "..."],
    "recommendations": ["...", "..."]
}}"""

    response = llm.query(
        summary_prompt,
        system_prompt="You are a senior market intelligence analyst creating executive reports."
    )

    try:
        report_data = json.loads(response)
    except json.JSONDecodeError:
        report_data = {
            "executive_summary": response[:500],
            "key_trends": [],
            "pain_points": [],
            "opportunities": [],
            "recommendations": []
        }

    # Add metadata
    report = {
        "report_type": "data_analysis",
        "generated_at": datetime.now().isoformat(),
        "data_sources": sources,
        "total_records": len(data),
        "report": report_data
    }

    return report


def generate_report_from_agents(topic: str, output_path: Path) -> dict:
    """Generate report using multi-agent system"""
    print(f"■ Using multi-agent system for topic: {topic}")

    vector_config = VectorDBConfig(
        persist_directory=settings.vectordb.PERSIST_DIR,
        collection_name=settings.vectordb.COLLECTION_NAME,
        embedding_model=settings.model.EMBEDDING_MODEL
    )

    orchestrator = Orchestrator(vector_db_config=vector_config)

    result = orchestrator.run_full_pipeline(
        topic=topic,
        generate_ideas=True,
        evaluate_ideas=True,
        num_ideas=5,
        top_k_research=10
    )

    # Convert to report format
    report = {
        "report_type": "multi_agent",
        "generated_at": datetime.now().isoformat(),
        "topic": topic,
        "execution_time": result.execution_time,
        "research": {
            "summary": result.research_results.summary if result.research_results else "",
            "key_findings": result.research_results.key_findings if result.research_results else [],
            "sources": result.research_results.sources if result.research_results else []
        } if result.research_results else None,
        "market_analysis": {
            "market_size": result.market_analysis.market_size if result.market_analysis else None,
            "market_maturity": result.market_analysis.market_maturity if result.market_analysis else None,
            "trends": result.market_analysis.trends if result.market_analysis else [],
            "opportunities": result.market_analysis.opportunities if result.market_analysis else [],
            "threats": result.market_analysis.threats if result.market_analysis else []
        } if result.market_analysis else None,
        "generated_ideas": [
            {
                "title": idea.title,
                "description": idea.description,
                "target_audience": idea.target_audience,
                "value_proposition": idea.value_proposition,
                "key_features": idea.key_features,
                "market_opportunity": idea.market_opportunity
            }
            for idea in (result.generated_ideas or [])
        ],
        "evaluations": [
            {
                "idea": eval_result.idea[:100] + "..." if len(eval_result.idea) > 100 else eval_result.idea,
                "overall_score": eval_result.overall_score,
                "feasibility_score": eval_result.feasibility_score,
                "market_potential_score": eval_result.market_potential_score,
                "innovation_score": eval_result.innovation_score,
                "verdict": eval_result.verdict,
                "strengths": eval_result.strengths,
                "weaknesses": eval_result.weaknesses,
                "recommendations": eval_result.recommendations
            }
            for eval_result in (result.evaluations or [])
        ],
        "top_ideas": [
            {
                "title": idea.title,
                "description": idea.description,
                "value_proposition": idea.value_proposition
            }
            for idea in (result.top_ideas or [])
        ],
        "summary": result.summary
    }

    return report


def generate_report_from_rag(topic: str, output_path: Path) -> dict:
    """Generate report using RAG insight generator"""
    print(f"■ Using RAG system for topic: {topic}")

    vector_config = VectorDBConfig(
```

```python
        persist_directory=settings.vectordb.PERSIST_DIR,
        collection_name=settings.vectordb.COLLECTION_NAME,
        embedding_model=settings.model.EMBEDDING_MODEL
    )

    retriever = RAGRetriever(vector_config)
    insight_generator = InsightGenerator()

    # Retrieve relevant documents
    documents = retriever.retrieve(topic, top_k=10)

    # Format context
    context = "\n\n".join([
        f"Document {i+1} (Score: {doc.score:.3f}):\n{doc.text[:500]}..."
        for i, doc in enumerate(documents)
    ])

    # Generate insights
    insights = insight_generator.generate(topic, context)

    # Create report
    report = {
        "report_type": "rag_insights",
        "generated_at": datetime.now().isoformat(),
        "topic": topic,
        "documents_analyzed": len(documents),
        "insights": {
            "summary": insights.summary,
            "pain_points": insights.pain_points,
            "opportunities": insights.opportunities,
            "signals": insights.signals
        },
        "sources": list(set([doc.metadata.get("source", "unknown") for doc in documents]))
    }

    return report


def save_report(report: dict, output_path: Path):
    """Save report to file"""
    output_path.parent.mkdir(parents=True, exist_ok=True)

    with open(output_path, "w", encoding="utf-8") as f:
        json.dump(report, f, ensure_ascii=False, indent=2)

    print(f"\n■ Report saved to: {output_path}")


def print_report_summary(report: dict):
    """Print a summary of the report"""
    print("\n" + "=" * 60)
    print("Report Summary")
    print("=" * 60)

    report_type = report.get("report_type", "unknown")
    print(f"\n■ Report Type: {report_type}")
    print(f"■ Generated: {report.get('generated_at', 'Unknown')}")

    if report_type == "data_analysis":
        print(f"■ Records Analyzed: {report.get('total_records', 0)}")
        report_data = report.get("report", {})
        print(f"■ Trends Identified: {len(report_data.get('key_trends', []))}")
        print(f"■ Opportunities: {len(report_data.get('opportunities', []))}")

    elif report_type == "multi_agent":
        print(f"■■  Execution Time: {report.get('execution_time', 0):.2f}s")
        if report.get("research"):
            print(f"■ Research Findings: {len(report['research'].get('key_findings', []))}")
        if report.get("market_analysis"):
            print(f"■ Market Maturity: {report['market_analysis'].get('market_maturity', 'Unknown')}")
        print(f"■ Ideas Generated: {len(report.get('generated_ideas', []))}")
        print(f"■ Top Ideas: {len(report.get('top_ideas', []))}")

    elif report_type == "rag_insights":
        print(f"■ Documents Analyzed: {report.get('documents_analyzed', 0)}")
        insights = report.get("insights", {})
        print(f"■ Pain Points: {len(insights.get('pain_points', []))}")
        print(f"■ Opportunities: {len(insights.get('opportunities', []))}")
        print(f"■ Signals: {len(insights.get('signals', []))}")


def main():
    """Main function"""
    parser = argparse.ArgumentParser(description="Generate market intelligence reports")
    parser.add_argument(
        "--method",
        choices=["data", "agents", "rag"],
        default="data",
        help="Report generation method (default: data)"
    )
    parser.add_argument(
        "--topic",
        type=str,
        help="Topic for agent-based or RAG-based reports (required for agents/rag methods)"
    )
    parser.add_argument(
        "--output",
        type=str,
        help="Output file path (default: reports/report_YYYYMMDD_HHMMSS.json)"
    )

    args = parser.parse_args()

    # Validate arguments
    if args.method in ["agents", "rag"] and not args.topic:
        print(f"■ Error: --topic is required for method '{args.method}'")
        return 1

    # Determine output path
    if args.output:
        output_path = Path(args.output)
    else:
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
        reports_dir = Path(settings.paths.REPORTS)
        output_path = reports_dir / f"report_{timestamp}.json"

    print("=" * 60)
    print("Market Intelligence Report Generator")
    print("=" * 60)

    try:
        # Generate report based on method
        if args.method == "data":
            print("\n■ Generating report from processed data...")
            report = generate_report_from_data(output_path)

        elif args.method == "agents":
            report = generate_report_from_agents(args.topic, output_path)

        elif args.method == "rag":
            report = generate_report_from_rag(args.topic, output_path)

        # Save report
        save_report(report, output_path)

        # Print summary
        print_report_summary(report)

        return 0

    except FileNotFoundError as e:
        print(f"\n■ Error: {str(e)}")
        if args.method == "data":
            print("   Run 'python scripts/run_analysis.py' first to process data.")
        return 1
    except Exception as e:
        print(f"\n■ Error generating report: {str(e)}")
        import traceback
        traceback.print_exc()
        return 1


if __name__ == "__main__":
    exit(main())

```

run_agents.py

```python
"""
Example script demonstrating the Multi-Agent Market Intelligence System

Usage:
    python scripts/run_agents.py
"""

from app.agents import Orchestrator
from app.vector_db.config import VectorDBConfig
from app.config.settings import get_settings

settings = get_settings()


def main():
    """Run the multi-agent system"""

    # Initialize vector DB config
    vector_config = VectorDBConfig(
        persist_directory=settings.vectordb.PERSIST_DIR,
        collection_name=settings.vectordb.COLLECTION_NAME,
        embedding_model=settings.model.EMBEDDING_MODEL
    )

    # Create orchestrator
    orchestrator = Orchestrator(vector_db_config=vector_config)

    # Example: Research and generate ideas for a topic
    topic = "AI-powered productivity tools for remote teams"

    print("=" * 60)
    print("Multi-Agent Market Intelligence System")
    print("=" * 60)
    print(f"\nTopic: {topic}\n")

    # Run full pipeline
    result = orchestrator.run_full_pipeline(
        topic=topic,
        generate_ideas=True,
        evaluate_ideas=True,
        num_ideas=5,
        top_k_research=10
    )

    # Display results
    print("\n" + "=" * 60)
    print("RESULTS SUMMARY")
    print("=" * 60)

    if result.research_results:
        print(f"\n■ Research Results:")
        print(f"   Documents found: {len(result.research_results.documents)}")
        print(f"   Key findings: {len(result.research_results.key_findings)}")
        print(f"   Sources: {', '.join(result.research_results.sources[:3])}...")

    if result.market_analysis:
        print(f"\n■ Market Analysis:")
        print(f"   Market Maturity: {result.market_analysis.market_maturity}")
        print(f"   Market Size: {result.market_analysis.market_size}")
        print(f"   Opportunities: {len(result.market_analysis.opportunities)}")
        print(f"   Trends: {', '.join(result.market_analysis.trends[:3])}...")

    if result.generated_ideas:
        print(f"\n■ Generated Ideas ({len(result.generated_ideas)}):")
        for i, idea in enumerate(result.generated_ideas, 1):
            print(f"\n   {i}. {idea.title}")
            print(f"      {idea.description[:100]}...")
            print(f"      Target: {idea.target_audience}")

    if result.evaluations:
        print(f"\n■ Evaluations:")
        for i, eval_result in enumerate(result.evaluations, 1):
            print(f"   Idea {i}:")
            print(f"      Overall Score: {eval_result.overall_score:.2f}/1.0")
            print(f"      Verdict: {eval_result.verdict}")
            print(f"      Strengths: {len(eval_result.strengths)}")

    if result.top_ideas:
        print(f"\n■ Top {len(result.top_ideas)} Ideas:")
        for i, idea in enumerate(result.top_ideas, 1):
            eval_result = result.evaluations[i-1] if i <= len(result.evaluations) else None
            score = eval_result.overall_score if eval_result else 0.0
            print(f"\n   {i}. {idea.title} (Score: {score:.2f})")
            print(f"      {idea.value_proposition}")

    print(f"\n■■  Execution Time: {result.execution_time:.2f}s")
    print(f"■ Summary: {result.summary}")
    print("\n" + "=" * 60)


if __name__ == "__main__":
    main()
```

run_analysis.py

```python
"""
Script to run data analysis pipeline

This script:
1. Loads raw scraped data (Reddit and Hacker News)
2. Preprocesses and cleans the text
3. Generates embeddings
4. Runs LLM analysis to extract market insights
5. Saves processed data to data/processed/

Usage:
    python scripts/run_analysis.py
"""

import os
import sys
from pathlib import Path

# Add project root to path
project_root = Path(__file__).parent.parent
sys.path.insert(0, str(project_root))

from pipelines.data_pipeline import run_pipeline, load_json_records
from app.config.settings import get_settings

settings = get_settings()


def main():
    """Run the data analysis pipeline"""
    print("=" * 60)
    print("Data Analysis Pipeline")
    print("=" * 60)

    raw_path = Path(settings.paths.RAW_DATA)
    processed_path = Path(settings.paths.PROCESSED_DATA)

    # Check if raw data exists
    reddit_file = raw_path / "reddit.json"
    hn_file = raw_path / "hn.json"

    files_exist = []
    if reddit_file.exists():
        files_exist.append(("Reddit", reddit_file))
    if hn_file.exists():
        files_exist.append(("Hacker News", hn_file))

    if not files_exist:
        print(f"\n■ No raw data files found in {raw_path}")
        print("   Please run 'python scripts/run_scrapers.py' first to collect data.")
        return 1

    print(f"\n■ Found {len(files_exist)} data source(s):")
    for name, file_path in files_exist:
        try:
            records = load_json_records(file_path)
            print(f"   ■ {name}: {len(records)} records")
        except Exception as e:
            print(f"   ■ {name}: Error loading - {str(e)}")
            return 1

    # Check for OpenAI API key
    if not settings.OPENAI_API_KEY:
        print("\n■ OPENAI_API_KEY not found in settings")
        print("   Please set it in your .env file or environment variables.")
        return 1

    print(f"\n■ Configuration:")
    print(f"   Embedding Model: {settings.model.EMBEDDING_MODEL}")
    print(f"   Analysis Model: {settings.model_routing.ANALYSIS_MODEL}")
    print(f"   Temperature: {settings.model.TEMPERATURE}")
    print(f"   Max Tokens: {settings.model.MAX_TOKENS}")

    # Run pipeline
    try:
        print("\n■ Starting analysis pipeline...")
        print("   This may take a while depending on the amount of data...")

        output_file = run_pipeline()

        print("\n" + "=" * 60)
        print("Pipeline Completed Successfully!")
        print("=" * 60)
        print(f"\n■ Processed data saved to: {output_file}")

        # Show statistics
        try:
            import json
            with open(output_file, "r", encoding="utf-8") as f:
                processed_data = json.load(f)

            print(f"\n■ Statistics:")
            print(f"   Total records processed: {len(processed_data)}")

            # Count by source
            sources = {}
            for record in processed_data:
                source = record.get("source", "unknown")
                sources[source] = sources.get(source, 0) + 1

            print(f"   Sources:")
            for source, count in sources.items():
                print(f"      - {source}: {count} records")

            print(f"\n■ Next steps:")
            print(f"   1. Index data into vector DB (if not already done)")
            print(f"   2. Run 'python scripts/generate_report.py' to generate insights")
            print(f"   3. Or use the multi-agent system: 'python scripts/run_agents.py'")

        except Exception as e:
            print(f"   ■■  Could not load statistics: {str(e)}")

        return 0

    except FileNotFoundError as e:
        print(f"\n■ File not found: {str(e)}")
        print("   Make sure raw data files exist in data/raw/")
        return 1
    except Exception as e:
        print(f"\n■ Pipeline failed: {str(e)}")
        import traceback
        traceback.print_exc()
        return 1


if __name__ == "__main__":
    exit(main())

```

run_scrapers.py

```python
"""
Script to run data collection scrapers (Reddit and Hacker News)

Usage:
    python scripts/run_scrapers.py
"""

import os
import sys
from pathlib import Path

# Add project root to path
project_root = Path(__file__).parent.parent
sys.path.insert(0, str(project_root))

from app.data_collectors.reddit_scraper import scrape_reddit
from app.data_collectors.hn_scraper import scrape_hackernews
from app.config.settings import get_settings

settings = get_settings()


def main():
    """Run all scrapers"""
    print("=" * 60)
    print("Data Collection Scrapers")
    print("=" * 60)

    # Check for required environment variables for Reddit
    reddit_required = ["REDDIT_CLIENT_ID", "REDDIT_CLIENT_SECRET", "REDDIT_USER_AGENT"]
    reddit_missing = [var for var in reddit_required if not os.getenv(var)]

    if reddit_missing:
        print(f"\n■■  Warning: Missing Reddit environment variables: {', '.join(reddit_missing)}")
        print("   Reddit scraping will be skipped.")
        print("   Set these in your .env file or environment.")
        reddit_enabled = False
    else:
        reddit_enabled = True

    results = {}

    # Scrape Reddit
    if reddit_enabled:
        try:
            print("\n■ Scraping Reddit...")
            reddit_path = scrape_reddit(limit=settings.scraper.REDDIT_LIMIT)
            results["reddit"] = {
                "status": "success",
                "path": reddit_path
            }
            print(f"■ Reddit scraping completed")
        except Exception as e:
            print(f"■ Reddit scraping failed: {str(e)}")
            results["reddit"] = {
                "status": "failed",
                "error": str(e)
            }
    else:
        results["reddit"] = {
            "status": "skipped",
            "reason": "Missing environment variables"
        }

    # Scrape Hacker News
    try:
        print("\n■ Scraping Hacker News...")
        hn_path = scrape_hackernews(limit=settings.scraper.HN_LIMIT)
        results["hackernews"] = {
            "status": "success",
            "path": hn_path
        }
        print(f"■ Hacker News scraping completed")
    except Exception as e:
        print(f"■ Hacker News scraping failed: {str(e)}")
        results["hackernews"] = {
            "status": "failed",
            "error": str(e)
        }

    # Summary
    print("\n" + "=" * 60)
    print("Scraping Summary")
    print("=" * 60)

    for source, result in results.items():
        status_icon = "■" if result["status"] == "success" else "■" if result["status"] == "failed" else "■■"
        print(f"{status_icon} {source.capitalize()}: {result['status']}")
        if result["status"] == "success":
            print(f"   Path: {result['path']}")
        elif result["status"] == "failed":
            print(f"   Error: {result.get('error', 'Unknown error')}")

    # Check if we have data for analysis
    success_count = sum(1 for r in results.values() if r["status"] == "success")
    if success_count > 0:
        print(f"\n■ Successfully scraped {success_count} source(s)")
        print("   Next step: Run 'python scripts/run_analysis.py' to process the data")
    else:
        print("\n■■  No data was successfully scraped. Please check errors above.")
        return 1

    return 0


if __name__ == "__main__":
    exit(main())
```