



## \_\_init\_\_.py

```
1 | """Multi-Agent Market Intelligence System"""
2 |
3 | from app.agents.base_agent import BaseAgent
4 | from app.agents.researcher_agent import ResearcherAgent
5 | from app.agents.market_agent import MarketAgent
6 | from app.agents.evaluator_agent import EvaluatorAgent
7 | from app.agents.idea_generator_agent import IdeaGeneratorAgent
8 | from app.agents.orchestrator import Orchestrator
9 | from app.agents.schemas import (
10 |     AgentRole,
11 |     TaskStatus,
12 |     ResearchResult,
13 |     MarketAnalysis,
14 |     GeneratedIdea,
15 |     IdeaEvaluation,
16 |     OrchestrationResult,
17 |     AgentTask,
18 |     AgentMessage,
19 | )
20 |
21 | __all__ = [
22 |     "BaseAgent",
23 |     "ResearcherAgent",
24 |     "MarketAgent",
25 |     "EvaluatorAgent",
26 |     "IdeaGeneratorAgent",
27 |     "Orchestrator",
28 |     "AgentRole",
29 |     "TaskStatus",
30 |     "ResearchResult",
31 |     "MarketAnalysis",
32 |     "GeneratedIdea",
33 |     "IdeaEvaluation",
34 |     "OrchestrationResult",
35 |     "AgentTask",
36 |     "AgentMessage",
37 | ]
38 |
```

## base\_agent.py

```
1 | from abc import ABC, abstractmethod
2 | from typing import Any, Dict, Optional
3 | from datetime import datetime
4 | import uuid
5 |
6 | from app.llm.client import LLMWrapper
7 | from app.llm.model_router import ModelRouter
8 | from app.agents.schemas import AgentRole, TaskStatus, AgentTask
9 |
10 |
11 | class BaseAgent(ABC):
12 |
13 |     def __init__(
14 |         self,
15 |         role: AgentRole,
16 |         model: Optional[str] = None,
17 |         system_prompt: Optional[str] = None
18 |     ):
19 |         self.role = role
20 |         self.model = model or ModelRouter.get_model_for_task("synthesis")
21 |         self.llm = LLMWrapper(model=self.model)
22 |         self.system_prompt = system_prompt or self._get_default_system_prompt()
23 |         self.task_history: list[AgentTask] = []
24 |
25 |     @abstractmethod
26 |     def _get_default_system_prompt(self) -> str:
27 |         pass
28 |
29 |     @abstractmethod
30 |     def execute(self, task: AgentTask) -> AgentTask:
31 |         pass
32 |
33 |     def create_task(
34 |         self,
35 |         task_type: str,
36 |         input_data: Dict[str, Any]
37 |     ) -> AgentTask:
38 |         task = AgentTask(
39 |             task_id=str(uuid.uuid4()),
40 |             agent_role=self.role,
41 |             task_type=task_type,
42 |             input_data=input_data,
43 |             status=TaskStatus.PENDING,
44 |             created_at=datetime.now()
45 |         )
46 |         return task
47 |
48 |     def _update_task_status(
49 |         self,
50 |         task: AgentTask,
51 |         status: TaskStatus,
52 |         result: Optional[Any] = None,
53 |         error: Optional[str] = None
54 |     ) -> AgentTask:
55 |         task.status = status
56 |         if result:
57 |             task.result = result
58 |         if error:
59 |             task.error = error
60 |         if status == TaskStatus.COMPLETED:
61 |             task.completed_at = datetime.now()
62 |         return task
63 |
64 |     def query_llm(
65 |         self,
66 |         prompt: str,
67 |         system_prompt: Optional[str] = None,
68 |         temperature: Optional[float] = None
69 |     ) -> str:
70 |         sys_prompt = system_prompt or self.system_prompt
71 |         return self.llm.query(prompt, system_prompt=sys_prompt)
72 |
73 |     def log_task(self, task: AgentTask):
74 |         self.task_history.append(task)
75 |
76 |     def get_task_history(self) -> list[AgentTask]:
77 |         return self.task_history
78 |
79 |     def __repr__(self) -> str:
80 |         return f"{self.__class__.__name__}(role={self.role.value})"
81 |
```

## evaluator\_agent.py

```
1 | import json
2 | from typing import List
3 | from datetime import datetime
4 |
5 | from app.agents.base_agent import BaseAgent
6 | from app.agents.schemas import (
7 |     AgentRole, AgentTask, TaskStatus, IdeaEvaluation,
8 |     GeneratedIdea, MarketAnalysis
9 | )
10 | from app.llm.model_router import ModelRouter
11 |
12 |
13 | class EvaluatorAgent(BaseAgent):
14 |
15 |     def __init__(self):
16 |         super().__init__(
17 |             role=AgentRole.EVALUATOR,
18 |             model=ModelRouter.get_model_for_task("extraction")
19 |         )
20 |
21 |     def _get_default_system_prompt(self) -> str:
22 |         return """You are a senior startup evaluator and advisor with experience in:
23 | - Product-market fit assessment
24 | - Market opportunity evaluation
25 | - Feasibility analysis
26 | - Risk assessment
27 | - Investment decision-making
28 |
29 | Provide honest, critical, and constructive evaluations. Be specific about strengths,
30 | weaknesses, and actionable recommendations."""
31 |
32 |     def execute(self, task: AgentTask) -> AgentTask:
33 |         """Execute an evaluation task"""
34 |         try:
35 |             self._update_task_status(task, TaskStatus.IN_PROGRESS)
36 |
37 |             idea = task.input_data.get("idea")
38 |             idea_obj = task.input_data.get("idea_obj") # GeneratedIdea object
39 |             market_analysis = task.input_data.get("market_analysis") # MarketAnalysis object
40 |
41 |             # Extract idea details
42 |             if idea_obj and isinstance(idea_obj, GeneratedIdea):
43 |                 idea_text = f"""
44 | Title: {idea_obj.title}
45 | Description: {idea_obj.description}
46 | Target Audience: {idea_obj.target_audience}
47 | Value Proposition: {idea_obj.value_proposition}
48 | Key Features: {'', '.join(idea_obj.key_features)}
49 | Market Opportunity: {idea_obj.market_opportunity}
50 | """
51 |             else:
52 |                 idea_text = idea or "No idea provided"
53 |
54 |             market_context = ""
55 |             if market_analysis and isinstance(market_analysis, MarketAnalysis):
56 |                 market_context = f"""
57 | Market Context:
58 | - Market Size: {market_analysis.market_size}
59 | - Market Maturity: {market_analysis.market_maturity}
60 | - Trends: {'', '.join(market_analysis.trends)}
61 | - Opportunities: {'', '.join(market_analysis.opportunities)}
62 | - Threats: {'', '.join(market_analysis.threats)}
63 | - Competitive Landscape: {market_analysis.competitive_landscape}
64 | """
65 |
66 |             evaluation_prompt = f"""Evaluate the following startup idea:
67 |
68 | {idea_text}
69 |
70 | {market_context if market_context else ""}
71 |
72 | Provide a comprehensive evaluation in JSON format:
73 | {{
74 |     "feasibility_score": 0.0-1.0,
75 |     "market_potential_score": 0.0-1.0,
76 |     "innovation_score": 0.0-1.0,
77 |     "overall_score": 0.0-1.0,
78 |     "strengths": ["strength 1", "strength 2"],
79 |     "weaknesses": ["weakness 1", "weakness 2"],
80 |     "risks": ["risk 1", "risk 2"],
81 |     "recommendations": ["recommendation 1", "recommendation 2"],
82 |     "verdict": "high_potential|medium_potential|low_potential"
83 | }}
84 |
85 | Scoring guidelines:
86 | - Feasibility: Can this be built? (technical, resource, time constraints)
87 | - Market Potential: Is there a real market need? (size, growth, willingness to pay)
88 | - Innovation: How novel/unique is this? (differentiation, competitive advantage)
89 | - Overall: Weighted average considering all factors
90 |
91 | Be honest and specific."""
92 |
93 |             response = self.query_llm(evaluation_prompt)
94 |
95 |             try:
96 |                 parsed = json.loads(response)
```

```

97         except json.JSONDecodeError:
98             parsed = self._parse_evaluation_fallback(response)
99
100         overall_score = parsed.get("overall_score")
101         if overall_score is None:
102             overall_score = (
103                 parsed.get("feasibility_score", 0.5) * 0.3 +
104                 parsed.get("market_potential_score", 0.5) * 0.4 +
105                 parsed.get("innovation_score", 0.5) * 0.3
106             )
107
108         result = IdeaEvaluation(
109             idea=idea_text,
110             feasibility_score=float(parsed.get("feasibility_score", 0.5)),
111             market_potential_score=float(parsed.get("market_potential_score", 0.5)),
112             innovation_score=float(parsed.get("innovation_score", 0.5)),
113             overall_score=float(overall_score),
114             strengths=parsed.get("strengths", []),
115             weaknesses=parsed.get("weaknesses", []),
116             risks=parsed.get("risks", []),
117             recommendations=parsed.get("recommendations", []),
118             verdict=parsed.get("verdict", "medium_potential")
119         )
120
121         self._update_task_status(task, TaskStatus.COMPLETED, result=result)
122         self.log_task(task)
123
124     except Exception as e:
125         self._update_task_status(
126             task,
127             TaskStatus.FAILED,
128             error=str(e)
129         )
130         self.log_task(task)
131
132     return task
133
134 def _parse_evaluation_fallback(self, text: str) -> dict:
135     """Fallback parser if JSON parsing fails"""
136     return {
137         "feasibility_score": 0.5,
138         "market_potential_score": 0.5,
139         "innovation_score": 0.5,
140         "overall_score": 0.5,
141         "strengths": [],
142         "weaknesses": [],
143         "risks": [],
144         "recommendations": [],
145         "verdict": "medium_potential"
146     }
147
148 def evaluate_idea(
149     self,
150     idea: str = None,
151     idea_obj: GeneratedIdea = None,
152     market_analysis: MarketAnalysis = None
153 ) -> IdeaEvaluation:
154     """Convenience method for quick idea evaluation"""
155     task = self.create_task(
156         "idea_evaluation",
157         {
158             "idea": idea,
159             "idea_obj": idea_obj,
160             "market_analysis": market_analysis
161         }
162     )
163     task = self.execute(task)
164     return task.result
165

```

## idea\_generator\_agent.py

```
1 | import json
2 | from typing import List
3 | from datetime import datetime
4 |
5 | from app.agents.base_agent import BaseAgent
6 | from app.agents.schemas import (
7 |     AgentRole, AgentTask, TaskStatus, GeneratedIdea,
8 |     ResearchResult, MarketAnalysis
9 | )
10 | from app.llm.model_router import ModelRouter
11 |
12 |
13 | class IdeaGeneratorAgent(BaseAgent):
14 |     """Agent responsible for generating startup ideas based on research"""
15 |
16 |     def __init__(self):
17 |         super().__init__(
18 |             role=AgentRole.IDEA_GENERATOR,
19 |             model=ModelRouter.get_model_for_task("synthesis")
20 |         )
21 |
22 |     def _get_default_system_prompt(self) -> str:
23 |         return """You are a creative startup ideation expert with deep knowledge of:
24 | - Emerging technologies and trends
25 | - Market gaps and opportunities
26 | - User pain points and unmet needs
27 | - Successful startup patterns
28 |
29 | Generate innovative, feasible, and market-driven startup ideas. Focus on:
30 | - Solving real problems
31 | - Clear value propositions
32 | - Specific target audiences
33 | - Differentiated features"""
34 |
35 |     def execute(self, task: AgentTask) -> AgentTask:
36 |         """Execute an idea generation task"""
37 |         try:
38 |             self.update_task_status(task, TaskStatus.IN_PROGRESS)
39 |
40 |             topic = task.input_data.get("topic", "")
41 |             num_ideas = task.input_data.get("num_ideas", 3)
42 |             research_result = task.input_data.get("research_result")
43 |             market_analysis = task.input_data.get("market_analysis")
44 |
45 |             # Build context from research and market analysis
46 |             context_parts = []
47 |
48 |             if research_result and isinstance(research_result, ResearchResult):
49 |                 context_parts.append(f"""
50 | Research Findings:
51 | {research_result.summary}
52 |
53 | Key Insights:
54 | {chr(10).join(f"- {finding}" for finding in research_result.key_findings)}
55 | """)
56 |
57 |             if market_analysis and isinstance(market_analysis, MarketAnalysis):
58 |                 context_parts.append(f"""
59 | Market Analysis:
60 | - Market Size: {market_analysis.market_size}
61 | - Maturity: {market_analysis.market_maturity}
62 | - Trends: {'', '.join(market_analysis.trends)}
63 | - Opportunities: {'', '.join(market_analysis.opportunities)}
64 | - Threats: {'', '.join(market_analysis.threats)}
65 | """)
66 |
67 |             context = "\n".join(context_parts) if context_parts else ""
68 |
69 |             # Generate ideas
70 |             generation_prompt = f"""Generate {num_ideas} innovative startup ideas related to: {topic}
71 |
72 | {context if context else "Base ideas on general market knowledge and trends."}
73 |
74 | For each idea, provide JSON format:
75 | {{
76 |     "title": "Short, catchy title",
77 |     "description": "2-3 sentence description",
78 |     "target_audience": "Specific target audience",
79 |     "value_proposition": "Clear value proposition",
80 |     "key_features": ["feature 1", "feature 2", "feature 3"],
81 |     "market_opportunity": "Why this is a good opportunity",
82 |     "inspiration_sources": ["source 1", "source 2"]
83 | }}
84 |
85 | Return as a JSON array of ideas. Make each idea:
86 | - Specific and actionable
87 | - Based on real pain points or opportunities
88 | - Technically feasible
89 | - Differentiated from existing solutions"""
90 |
91 |             response = self.query_llm(generation_prompt, temperature=0.8)
92 |
93 |             # Parse response
94 |             try:
95 |                 parsed = json.loads(response)
96 |                 # Handle both array and single object responses
```

```

97         if isinstance(parsed, dict):
98             parsed = [parsed]
99     except json.JSONDecodeError:
100         # Try to extract ideas from text
101         parsed = self._parse_ideas_fallback(response)
102
103     # Create GeneratedIdea objects
104     ideas = []
105     for idea_data in parsed[:num_ideas]:
106         idea = GeneratedIdea(
107             title=idea_data.get("title", "Untitled Idea"),
108             description=idea_data.get("description", ""),
109             target_audience=idea_data.get("target_audience", ""),
110             value_proposition=idea_data.get("value_proposition", ""),
111             key_features=idea_data.get("key_features", []),
112             market_opportunity=idea_data.get("market_opportunity", ""),
113             inspiration_sources=idea_data.get("inspiration_sources", [])
114         )
115         ideas.append(idea)
116
117     self._update_task_status(task, TaskStatus.COMPLETED, result=ideas)
118     self.log_task(task)
119
120 except Exception as e:
121     self._update_task_status(
122         task,
123         TaskStatus.FAILED,
124         error=str(e)
125     )
126     self.log_task(task)
127
128 return task
129
130 def _parse_ideas_fallback(self, text: str) -> List[dict]:
131     """Fallback parser if JSON parsing fails"""
132     # Simple extraction - look for numbered or bulleted ideas
133     ideas = []
134     lines = text.split("\n")
135     current_idea = {}
136
137     for line in lines:
138         line = line.strip()
139         if not line:
140             continue
141
142         if line.startswith(("1.", "2.", "3.", "4.", "5.)) or \
143             (line[0].isupper() and len(line) < 100 and ":" not in line):
144             if current_idea:
145                 ideas.append(current_idea)
146                 current_idea = {"title": line.lstrip("1234567890. ")}
147             elif ":" in line and current_idea:
148                 key, value = line.split(":", 1)
149                 key = key.lower().replace(" ", "_")
150                 current_idea[key] = value.strip()
151
152     if current_idea:
153         ideas.append(current_idea)
154
155     return ideas if ideas else [{"title": "Generated Idea", "description": text[:200]}]
156
157 def generate_ideas(
158     self,
159     topic: str,
160     num_ideas: int = 3,
161     research_result=None,
162     market_analysis=None
163 ) -> List[GeneratedIdea]:
164     """Convenience method for quick idea generation"""
165     task = self.create_task(
166         "idea_generation",
167         {
168             "topic": topic,
169             "num_ideas": num_ideas,
170             "research_result": research_result,
171             "market_analysis": market_analysis
172         }
173     )
174     task = self.execute(task)
175     return task.result if task.result else []
176

```

## market\_agent.py

```
1 | import json
2 | from typing import List, Optional
3 | from datetime import datetime
4 |
5 | from app.agents.base_agent import BaseAgent
6 | from app.agents.schemas import AgentRole, AgentTask, TaskStatus, MarketAnalysis, ResearchResult
7 | from app.llm.model_router import ModelRouter
8 |
9 |
10 | class MarketAgent(BaseAgent):
11 |
12 |     def __init__(self):
13 |         super().__init__(
14 |             role=AgentRole.MARKET_ANALYST,
15 |             model=ModelRouter.get_model_for_task("extraction")
16 |         )
17 |
18 |     def _get_default_system_prompt(self) -> str:
19 |         return """You are a senior market analyst with expertise in startup ecosystems,
20 | technology markets, and emerging trends. Your role is to:
21 | - Analyze market dynamics and trends
22 | - Assess market size and growth potential
23 | - Identify opportunities and threats
24 | - Evaluate competitive landscapes
25 | - Determine market maturity stages
26 | Provide data-driven, objective analysis."""
27 |
28 |     def execute(self, task: AgentTask) -> AgentTask:
29 |         """Execute a market analysis task"""
30 |         try:
31 |             self._update_task_status(task, TaskStatus.IN_PROGRESS)
32 |
33 |             topic = task.input_data.get("topic", "")
34 |             research_result = task.input_data.get("research_result")
35 |
36 |             context = ""
37 |             if research_result and isinstance(research_result, ResearchResult):
38 |                 context = f"""
39 | Research Summary:
40 | {research_result.summary}
41 |
42 | Key Findings:
43 | {chr(10).join(f"- {finding}" for finding in research_result.key_findings)}
44 |
45 | Sources: {' '.join(research_result.sources)}
46 | """
47 |
48 |             analysis_prompt = f"""Analyze the market for: {topic}
49 |
50 | {context if context else "No prior research provided. Base your analysis on general market knowledge."}
51 |
52 | Provide a comprehensive market analysis in JSON format:
53 | {{
54 |     "market_size": "Estimated market size (e.g., '$X billion', 'growing rapidly', 'unknown')",
55 |     "trends": ["trend 1", "trend 2", "trend 3"],
56 |     "opportunities": ["opportunity 1", "opportunity 2"],
57 |     "threats": ["threat 1", "threat 2"],
58 |     "competitive_landscape": "Description of competition and market structure",
59 |     "market_maturity": "emerging|growing|mature|declining",
60 |     "confidence_score": 0.0-1.0
61 | }}
62 |
63 | Be specific and data-driven where possible."""
64 |
65 |             response = self.query_llm(analysis_prompt)
66 |
67 |             try:
68 |                 parsed = json.loads(response)
69 |             except json.JSONDecodeError:
70 |                 parsed = self._parse_market_analysis_fallback(response)
71 |
72 |             result = MarketAnalysis(
73 |                 topic=topic,
74 |                 market_size=parsed.get("market_size"),
75 |                 trends=parsed.get("trends", []),
76 |                 opportunities=parsed.get("opportunities", []),
77 |                 threats=parsed.get("threats", []),
78 |                 competitive_landscape=parsed.get("competitive_landscape", ""),
79 |                 market_maturity=parsed.get("market_maturity", "unknown"),
80 |                 confidence_score=float(parsed.get("confidence_score", 0.5))
81 |             )
82 |
83 |             self._update_task_status(task, TaskStatus.COMPLETED, result=result)
84 |             self.log_task(task)
85 |
86 |         except Exception as e:
87 |             self._update_task_status(
88 |                 task,
89 |                 TaskStatus.FAILED,
90 |                 error=str(e)
91 |             )
92 |             self.log_task(task)
93 |
94 |         return task
95 |
96 |     def _parse_market_analysis_fallback(self, text: str) -> dict:
```



```

97 |         """Fallback parser if JSON parsing fails"""
98 |         return {
99 |             "market_size": "Unknown",
100 |             "trends": [],
101 |             "opportunities": [],
102 |             "threats": [],
103 |             "competitive_landscape": text[:500],
104 |             "market_maturity": "unknown",
105 |             "confidence_score": 0.5
106 |         }
107 |
108 |     def analyze_market(
109 |         self,
110 |         topic: str,
111 |         research_result: Optional[ResearchResult] = None
112 |     ) -> MarketAnalysis:
113 |         """Convenience method for quick market analysis"""
114 |         task = self.create_task(
115 |             "market_analysis",
116 |             {
117 |                 "topic": topic,
118 |                 "research_result": research_result
119 |             }
120 |         )
121 |         task = self.execute(task)
122 |         return task.result
123 |

```

## orchestrator.py

```
1 import time
2 from typing import List, Optional, Dict, Any
3 from datetime import datetime
4
5 from app.agents.base_agent import BaseAgent
6 from app.agents.researcher_agent import ResearcherAgent
7 from app.agents.market_agent import MarketAgent
8 from app.agents.evaluator_agent import EvaluatorAgent
9 from app.agents.idea_generator_agent import IdeaGeneratorAgent
10 from app.agents.schemas import (
11     AgentRole, AgentTask, TaskStatus, OrchestrationResult,
12     ResearchResult, MarketAnalysis, GeneratedIdea, IdeaEvaluation
13 )
14 from app.vector_db.config import VectorDBConfig
15
16
17 class Orchestrator:
18     """Orchestrates the multi-agent system to complete complex tasks"""
19
20     def __init__(self, vector_db_config: VectorDBConfig):
21         self.researcher = ResearcherAgent(vector_db_config)
22         self.market_agent = MarketAgent()
23         self.evaluator = EvaluatorAgent()
24         self.idea_generator = IdeaGeneratorAgent()
25         self.execution_history: List[OrchestrationResult] = []
26
27     def run_full_pipeline(
28         self,
29         topic: str,
30         generate_ideas: bool = True,
31         evaluate_ideas: bool = True,
32         num_ideas: int = 5,
33         top_k_research: int = 10
34     ) -> OrchestrationResult:
35         """
36         Run the complete pipeline:
37         1. Research the topic
38         2. Analyze the market
39         3. Generate ideas (if requested)
40         4. Evaluate ideas (if requested)
41         5. Return top ideas
42
43         Args:
44             topic: The topic to research and generate ideas for
45             generate_ideas: Whether to generate ideas
46             evaluate_ideas: Whether to evaluate generated ideas
47             num_ideas: Number of ideas to generate
48             top_k_research: Number of research documents to retrieve
49
50         Returns:
51             OrchestrationResult with all findings
52         """
53         start_time = time.time()
54         result = OrchestrationResult(
55             task_id=f"orchestration_{datetime.now().strftime('%Y%m%d_%H%M%S')}",
56             status=TaskStatus.IN_PROGRESS
57         )
58
59         try:
60             print(f"■ Researching: {topic}")
61             research_task = self.researcher.create_task(
62                 "research",
63                 {"query": topic, "top_k": top_k_research}
64             )
65             research_task = self.researcher.execute(research_task)
66
67             if research_task.status == TaskStatus.COMPLETED:
68                 result.research_results = research_task.result
69                 print(f"■ Research completed: {len(research_task.result.documents)} documents found")
70             else:
71                 print(f"■ Research failed: {research_task.error}")
72                 result.status = TaskStatus.FAILED
73                 return result
74
75             print(f"■ Analyzing market: {topic}")
76             market_task = self.market_agent.create_task(
77                 "market_analysis",
78                 {
79                     "topic": topic,
80                     "research_result": result.research_results
81                 }
82             )
83             market_task = self.market_agent.execute(market_task)
84
85             if market_task.status == TaskStatus.COMPLETED:
86                 result.market_analysis = market_task.result
87                 print(f"■ Market analysis completed: {market_task.result.market_maturity} market")
88             else:
89                 print(f"■ Market analysis failed: {market_task.error}")
90
91             if generate_ideas:
92                 print(f"■ Generating {num_ideas} ideas...")
93                 idea_task = self.idea_generator.create_task(
94                     "idea_generation",
95                     {
96                         "topic": topic,
```

```

97         "num_ideas": num_ideas,
98         "research_result": result.research_results,
99         "market_analysis": result.market_analysis
100     }
101 )
102 idea_task = self.idea_generator.execute(idea_task)
103
104 if idea_task.status == TaskStatus.COMPLETED:
105     result.generated_ideas = idea_task.result
106     print(f"■ Generated {len(idea_task.result)} ideas")
107 else:
108     print(f"■ Idea generation failed: {idea_task.error}")
109
110 if evaluate_ideas and result.generated_ideas:
111     print(f"■ Evaluating {len(result.generated_ideas)} ideas...")
112     evaluations = []
113
114     for idea in result.generated_ideas:
115         eval_task = self.evaluator.create_task(
116             "idea_evaluation",
117             {
118                 "idea_obj": idea,
119                 "market_analysis": result.market_analysis
120             }
121         )
122         eval_task = self.evaluator.execute(eval_task)
123
124         if eval_task.status == TaskStatus.COMPLETED:
125             evaluations.append(eval_task.result)
126         else:
127             print(f"■ Evaluation failed for idea: {idea.title}")
128
129     result.evaluations = evaluations
130     print(f"■ Evaluated {len(evaluations)} ideas")
131
132     if evaluations:
133         result.top_ideas = self._rank_ideas(
134             result.generated_ideas,
135             evaluations
136         )
137         print(f"■ Top {len(result.top_ideas)} ideas selected")
138
139     result.summary = self._generate_summary(result)
140     result.execution_time = time.time() - start_time
141     result.status = TaskStatus.COMPLETED
142
143     print(f"\n■ Pipeline completed in {result.execution_time:.2f}s")
144
145 except Exception as e:
146     result.status = TaskStatus.FAILED
147     result.summary = f"Pipeline failed: {str(e)}"
148     result.execution_time = time.time() - start_time
149     print(f"■ Pipeline failed: {str(e)}")
150
151 self.execution_history.append(result)
152 return result
153
154 def _rank_ideas(
155     self,
156     ideas: List[GeneratedIdea],
157     evaluations: List[IdeaEvaluation]
158 ) -> List[GeneratedIdea]:
159     """Rank ideas based on evaluations and return top ones"""
160     if len(ideas) != len(evaluations):
161         return ideas[:3]
162
163     idea_eval_pairs = list(zip(ideas, evaluations))
164     idea_eval_pairs.sort(key=lambda x: x[1].overall_score, reverse=True)
165
166     return [idea for idea, _ in idea_eval_pairs[:3]]
167
168 def _generate_summary(self, result: OrchestrationResult) -> str:
169     """Generate a summary of the orchestration result"""
170     summary_parts = []
171
172     if result.research_results:
173         summary_parts.append(
174             f"Research: Found {len(result.research_results.documents)} relevant documents "
175             f"with {len(result.research_results.key_findings)} key findings."
176         )
177
178     if result.market_analysis:
179         summary_parts.append(
180             f"Market: {result.market_analysis.market_maturity} market with "
181             f"{len(result.market_analysis.opportunities)} opportunities identified."
182         )
183
184     if result.generated_ideas:
185         summary_parts.append(
186             f"Ideas: Generated {len(result.generated_ideas)} startup ideas."
187         )
188
189     if result.evaluations:
190         avg_score = sum(e.overall_score for e in result.evaluations) / len(result.evaluations)
191         summary_parts.append(
192             f"Evaluation: Average score {avg_score:.2f}/1.0 across {len(result.evaluations)} ideas."
193         )
194

```

```
195 |         if result.top_ideas:
196 |             summary_parts.append(
197 |                 f"Top Ideas: Selected {len(result.top_ideas)} highest-potential ideas."
198 |             )
199 |
200 |         return " ".join(summary_parts)
201 |
202 | def get_execution_history(self) -> List[OrchestrationResult]:
203 |     """Get execution history"""
204 |     return self.execution_history
205 |
206 | def __repr__(self) -> str:
207 |     return f"Orchestrator(agents=[Researcher, Market, Evaluator, IdeaGenerator])"
208 |
```

## researcher\_agent.py

```
1 | import json
2 | from typing import List, Dict, Any
3 | from datetime import datetime
4 |
5 | from app.agents.base_agent import BaseAgent
6 | from app.agents.schemas import AgentRole, AgentTask, TaskStatus, ResearchResult
7 | from app.rag.retriever import RAGRetriever
8 | from app.vector_db.config import VectorDBConfig
9 |
10 |
11 | class ResearcherAgent(BaseAgent):
12 |
13 |     def __init__(self, vector_db_config: VectorDBConfig):
14 |         super().__init__(role=AgentRole.RESEARCHER)
15 |         self.retriever = RAGRetriever(vector_db_config)
16 |
17 |     def _get_default_system_prompt(self) -> str:
18 |         return """You are a senior research analyst specializing in startup and technology markets.
19 | Your role is to:
20 | - Extract key insights from research documents
21 | - Identify patterns and trends
22 | - Summarize findings clearly and concisely
23 | - Highlight the most important information"""
24 |
25 |     def execute(self, task: AgentTask) -> AgentTask:
26 |         try:
27 |             self._update_task_status(task, TaskStatus.IN_PROGRESS)
28 |
29 |             query = task.input_data.get("query", "")
30 |             top_k = task.input_data.get("top_k", 10)
31 |
32 |             documents = self.retriever.retrieve(query, top_k=top_k)
33 |
34 |             context = self._format_documents(documents)
35 |
36 |             summary_prompt = f"""Based on the following research documents, provide:
37 | 1. A comprehensive summary (2-3 paragraphs)
38 | 2. Key findings (bullet points)
39 |
40 | Query: {query}
41 |
42 | Documents:
43 | {context}
44 |
45 | Format your response as JSON:
46 | {{
47 |     "summary": "...",
48 |     "key_findings": ["...", "..."]
49 | }}"""
50 |
51 |             response = self.query_llm(summary_prompt)
52 |
53 |             try:
54 |                 parsed = json.loads(response)
55 |                 summary = parsed.get("summary", "")
56 |                 key_findings = parsed.get("key_findings", [])
57 |             except json.JSONDecodeError:
58 |                 summary = response
59 |                 key_findings = self._extract_findings_from_text(response)
60 |
61 |             sources = [doc.metadata.get("source", "unknown") for doc in documents]
62 |             sources = list(set(sources))
63 |
64 |             result = ResearchResult(
65 |                 query=query,
66 |                 documents=[
67 |                     {
68 |                         "id": doc.id,
69 |                         "text": doc.text,
70 |                         "metadata": doc.metadata,
71 |                         "score": doc.score
72 |                     }
73 |                     for doc in documents
74 |                 ],
75 |                 summary=summary,
76 |                 key_findings=key_findings,
77 |                 sources=sources,
78 |                 timestamp=datetime.now()
79 |             )
80 |
81 |             self._update_task_status(task, TaskStatus.COMPLETED, result=result)
82 |             self.log_task(task)
83 |
84 |         except Exception as e:
85 |             self._update_task_status(
86 |                 task,
87 |                 TaskStatus.FAILED,
88 |                 error=str(e)
89 |             )
90 |             self.log_task(task)
91 |
92 |         return task
93 |
94 |     def _format_documents(self, documents: List) -> str:
95 |         """Format documents for LLM context"""
96 |         formatted = []
```

```

97 |         for i, doc in enumerate(documents, 1):
98 |             formatted.append(
99 |                 f"Document {i} (Relevance: {doc.score:.3f}): \n"
100 |                 f"Source: {doc.metadata.get('source', 'unknown')} \n"
101 |                 f"Content: {doc.text[:500]}...\n"
102 |             )
103 |         return "\n".join(formatted)
104 |
105 | def _extract_findings_from_text(self, text: str) -> List[str]:
106 |     """Extract findings from text if JSON parsing fails"""
107 |     lines = text.split("\n")
108 |     findings = []
109 |     for line in lines:
110 |         line = line.strip()
111 |         if line and (line.startswith("-") or line.startswith("*") or
112 |                     (line[0].isdigit() and "." in line[:3])):
113 |             findings.append(line.lstrip("-* ").split(".", 1)[-1])
114 |     return findings[:10]
115 |
116 | def research(self, query: str, top_k: int = 10) -> ResearchResult:
117 |     """Convenience method for quick research"""
118 |     task = self.create_task("research", {"query": query, "top_k": top_k})
119 |     task = self.execute(task)
120 |     return task.result
121 |

```

## schemas.py

```
1 | from dataclasses import dataclass
2 | from typing import List, Dict, Any, Optional
3 | from datetime import datetime
4 | from enum import Enum
5 |
6 |
7 | class AgentRole(str, Enum):
8 |     RESEARCHER = "researcher"
9 |     MARKET_ANALYST = "market_analyst"
10 |     EVALUATOR = "evaluator"
11 |     IDEA_GENERATOR = "idea_generator"
12 |
13 |
14 | class TaskStatus(str, Enum):
15 |     PENDING = "pending"
16 |     IN_PROGRESS = "in_progress"
17 |     COMPLETED = "completed"
18 |     FAILED = "failed"
19 |
20 |
21 | @dataclass
22 | class ResearchResult:
23 |     query: str
24 |     documents: List[Dict[str, Any]]
25 |     summary: str
26 |     key_findings: List[str]
27 |     sources: List[str]
28 |     timestamp: datetime
29 |
30 |
31 | @dataclass
32 | class MarketAnalysis:
33 |     topic: str
34 |     market_size: Optional[str]
35 |     trends: List[str]
36 |     opportunities: List[str]
37 |     threats: List[str]
38 |     competitive_landscape: str
39 |     market_maturity: str
40 |     confidence_score: float
41 |
42 |
43 | @dataclass
44 | class IdeaEvaluation:
45 |     idea: str
46 |     feasibility_score: float
47 |     market_potential_score: float
48 |     innovation_score: float
49 |     overall_score: float
50 |     strengths: List[str]
51 |     weaknesses: List[str]
52 |     risks: List[str]
53 |     recommendations: List[str]
54 |     verdict: str
55 |
56 |
57 | @dataclass
58 | class GeneratedIdea:
59 |     title: str
60 |     description: str
61 |     target_audience: str
62 |     value_proposition: str
63 |     key_features: List[str]
64 |     market_opportunity: str
65 |     inspiration_sources: List[str]
66 |
67 |
68 | @dataclass
69 | class AgentTask:
70 |     task_id: str
71 |     agent_role: AgentRole
72 |     task_type: str
73 |     input_data: Dict[str, Any]
74 |     status: TaskStatus = TaskStatus.PENDING
75 |     result: Optional[Any] = None
76 |     error: Optional[str] = None
77 |     created_at: datetime = None
78 |     completed_at: Optional[datetime] = None
79 |
80 |
81 | @dataclass
82 | class AgentMessage:
83 |     from_agent: AgentRole
84 |     to_agent: AgentRole
85 |     message_type: str
86 |     content: Dict[str, Any]
87 |     timestamp: datetime
88 |
89 |
90 | @dataclass
91 | class OrchestrationResult:
92 |     task_id: str
93 |     research_results: Optional[ResearchResult] = None
94 |     market_analysis: Optional[MarketAnalysis] = None
95 |     generated_ideas: List[GeneratedIdea] = None
96 |     evaluations: List[IdeaEvaluation] = None
```

```
97 | top_ideas: List[GeneratedIdea] = None
98 | summary: str = ""
99 | execution_time: float = 0.0
100 | status: TaskStatus = TaskStatus.PENDING
101 |
```



## settings.py

```
1 | from functools import lru_cache
2 | from typing import Literal
3 |
4 | from pydantic_settings import BaseSettings, SettingsConfigDict
5 |
6 |
7 | # =====
8 | # Model Configuration
9 | # =====
10 | class ModelConfig(BaseSettings):
11 |     CHAT_MODEL: str = "gpt-4o-mini"
12 |     EMBEDDING_MODEL: str = "text-embedding-3-small"
13 |     TEMPERATURE: float = 0.3
14 |     MAX_TOKENS: int = 1000
15 |
16 |
17 | class ModelRoutingConfig(BaseSettings):
18 |     IDEATION_MODEL: str = "gpt-4o-mini"
19 |     ANALYSIS_MODEL: str = "gpt-4o"
20 |     SUMMARIZATION_MODEL: str = "gpt-4o-mini"
21 |
22 |
23 | # =====
24 | # Vector Database Config
25 | # =====
26 | class VectorDBConfig(BaseSettings):
27 |     PROVIDER: Literal["chroma", "pinecone", "qdrant"] = "chroma"
28 |     COLLECTION_NAME: str = "documents"
29 |     PERSIST_DIR: str = "./vector_store"
30 |
31 |
32 | # =====
33 | # Logging + Observability
34 | # =====
35 | class LoggingConfig(BaseSettings):
36 |     LOG_LEVEL: Literal["DEBUG", "INFO", "WARNING", "ERROR", "CRITICAL"] = "INFO"
37 |     LOG_FILE: str = "./logs/app.log"
38 |
39 |
40 | class ObservabilityConfig(BaseSettings):
41 |     ENABLE_TRACING: bool = True
42 |     TRACE_LLM_CALLS: bool = True
43 |     TRACE_PIPELINES: bool = True
44 |
45 |
46 | # =====
47 | # Scraper Config
48 | # =====
49 | class ScraperConfig(BaseSettings):
50 |     REDDIT_LIMIT: int = 100
51 |     HN_LIMIT: int = 100
52 |     X_LIMIT: int = 100
53 |     REQUEST_TIMEOUT: int = 10
54 |     RETRIES: int = 3
55 |     RATE_LIMIT_PER_MINUTE: int = 30
56 |     BACKOFF_FACTOR: float = 1.5
57 |
58 |
59 | # =====
60 | # LLM Control
61 | # =====
62 | class LLMConfig(BaseSettings):
63 |     MAX_RETRIES: int = 3
64 |     TIMEOUT: int = 60
65 |     COST_LIMIT_PER_RUN: float = 5.0
66 |
67 |
68 | # =====
69 | # Pipeline Config
70 | # =====
71 | class PipelineConfig(BaseSettings):
72 |     BATCH_SIZE: int = 32
73 |     EMBEDDING_BATCH_SIZE: int = 64
74 |     ENABLE_PARALLEL_STAGES: bool = True
75 |
76 |
77 | # =====
78 | # Runtime Flags
79 | # =====
80 | class RuntimeFlags(BaseSettings):
81 |     ENABLE_LLM: bool = True
82 |     ENABLE_SCRAPING: bool = True
83 |     DRY_RUN: bool = False
84 |
85 |
86 | # =====
87 | # Cache Config
88 | # =====
89 | class CacheConfig(BaseSettings):
90 |     ENABLE_CACHE: bool = True
91 |     CACHE_DIR: str = "./cache"
92 |
93 |
94 | # =====
95 | # Worker / Async Config
96 | # =====
```

```

97 | class WorkerConfig(BaseSettings):
98 |     MAX_CONCURRENT_TASKS: int = 5
99 |     QUEUE_SIZE: int = 100
100 |
101 |
102 | # =====
103 | # Paths Config
104 | # =====
105 | class Paths(BaseSettings):
106 |     RAW_DATA: str = "data/raw"
107 |     PROCESSED_DATA: str = "data/processed"
108 |     REPORTS: str = "reports"
109 |
110 |
111 | # =====
112 | # Root Application Settings
113 | # =====
114 | class AppSettings(BaseSettings):
115 |     model_config = SettingsConfigDict(env_file=".env")
116 |
117 |     OPENAI_API_KEY: str
118 |     PINECONE_API_KEY: str | None = None
119 |     ENV: Literal["development", "staging", "production"] = "development"
120 |
121 |     model: ModelConfig = ModelConfig()
122 |     model_routing: ModelRoutingConfig = ModelRoutingConfig()
123 |     vectordb: VectorDBConfig = VectorDBConfig()
124 |
125 |     logging: LoggingConfig = LoggingConfig()
126 |     observability: ObservabilityConfig = ObservabilityConfig()
127 |
128 |     scraper: ScraperConfig = ScraperConfig()
129 |     llm: LLMConfig = LLMConfig()
130 |     pipeline: PipelineConfig = PipelineConfig()
131 |
132 |     runtime_flags: RuntimeFlags = RuntimeFlags()
133 |     cache: CacheConfig = CacheConfig()
134 |     worker: WorkerConfig = WorkerConfig()
135 |     paths: Paths = Paths()
136 |
137 |
138 | # =====
139 | # Settings Factory (Singleton)
140 | # =====
141 | @lru_cache
142 | def get_settings() -> AppSettings:
143 |     return AppSettings()
144 |
145 |
146 | # default instance
147 | settings = get_settings()
148 |

```

## hn\_scraper.py

```
1 | import requests
2 | import json
3 | import os
4 | from datetime import datetime
5 | from app.config.settings import get_settings
6 |
7 | settings = get_settings()
8 |
9 | HN_API_URL = "http://hn.algolia.com/api/v1/search?tags=front_page"
10 |
11 | def scrape_hackernews(limit=100):
12 |     all_posts = []
13 |
14 |     response = requests.get(HN_API_URL)
15 |     response.raise_for_status()
16 |     data = response.json()
17 |
18 |     for post in data.get("hits", [])[:limit]:
19 |         all_posts.append({
20 |             "title": post.get("title"),
21 |             "body": post.get("story_text") or "",
22 |             "score": post.get("points"),
23 |             "comments": post.get("num_comments"),
24 |             "created_at": datetime.fromtimestamp(post.get("created_at_i")).isoformat()
25 |             if post.get("created_at_i") else None,
26 |             "source": "hackernews"
27 |         })
28 |
29 |     file_path = os.path.join(settings.paths.RAW_DATA, "hn.json")
30 |     with open(file_path, "w", encoding="utf-8") as f:
31 |         json.dump(all_posts, f, indent=2, ensure_ascii=False)
32 |
33 |     print(f"Scraped {len(all_posts)} Hacker News posts -> {file_path}")
34 |     return file_path
35 |
36 | if __name__ == "__main__":
37 |     scrape_hackernews(limit=settings.scrapers.HN_LIMIT)
38 |
```

## reddit\_scraper.py

```
1 | import json
2 | import os
3 | from datetime import datetime
4 | import praw
5 | from app.config.settings import get_settings
6 |
7 | settings = get_settings()
8 |
9 | SUBREDDITS = ['startups', 'Entrepreneur', 'SaaS', 'SideProject']
10 |
11 | def scrape_reddit(limit=100):
12 |     reddit = praw.Reddit(
13 |         client_id=os.getenv("REDDIT_CLIENT_ID"),
14 |         client_secret=os.getenv("REDDIT_CLIENT_SECRET"),
15 |         user_agent=os.getenv("REDDIT_USER_AGENT")
16 |     )
17 |
18 |     all_posts = []
19 |
20 |     for sub in SUBREDDITS:
21 |         subreddit = reddit.subreddit(sub)
22 |         for post in subreddit.hot(limit=limit):
23 |             all_posts.append({
24 |                 "title": post.title,
25 |                 "body": post.selftext,
26 |                 "score": post.score,
27 |                 "comments": post.num_comments,
28 |                 "created_at": datetime.fromtimestamp(post.created_utc).isoformat(),
29 |                 "source": f"reddit/r/{sub}"
30 |             })
31 |
32 |     file_path = os.path.join(settings.paths.RAW_DATA, "reddit.json")
33 |     os.makedirs(os.path.dirname(file_path), exist_ok=True)
34 |     with open(file_path, "w", encoding="utf-8") as f:
35 |         json.dump(all_posts, f, ensure_ascii=False)
36 |
37 |     print(f"Scraped {len(all_posts)} Reddit posts -> {file_path}")
38 |     return file_path
39 |
40 | if __name__ == "__main__":
41 |     scrape_reddit(limit=settings.scraper.REDDIT_LIMIT)
42 |
```

## test\_json.py

```
1 | import json
2 | import os
3 | from app.config.settings import get_settings
4 |
5 | settings = get_settings()
6 |
7 | def test_json(file_name):
8 |     path = os.path.join(settings.paths.RAW_DATA, file_name)
9 |     try:
10 |         with open(path, "r", encoding="utf-8") as f:
11 |             data = json.load(f)
12 |             print(f"{file_name} is valid JSON with {len(data)} records")
13 |     except Exception as e:
14 |         print(f"{file_name} is invalid JSON! Error: {e}")
15 |
16 | if __name__ == "__main__":
17 |     test_json("reddit.json")
18 |     test_json("hn.json")
19 |
```

## cache\_manager.py

```
1 | import hashlib
2 | import json
3 | from app.config.settings import get_settings
4 |
5 | settings = get_settings()
6 |
7 | class CacheManager:
8 |     def __init__(self):
9 |         self.cache = {}
10 |
11 |     def _generate_key(self, model: str, messages: list) -> str:
12 |         data = f"{model}:{json.dumps(messages, sort_keys=True)}"
13 |         return hashlib.md5(data.encode()).hexdigest()
14 |
15 |     def get(self, model: str, messages: list):
16 |         key = self._generate_key(model, messages)
17 |         return self.cache.get(key)
18 |
19 |     def set(self, model: str, messages: list, response: str):
20 |         key = self._generate_key(model, messages)
21 |         if settings.cache.ENABLE_CACHE:
22 |             self.cache[key] = response
23 |
```

## client.py

```
1 | from app.config.settings import get_settings
2 | from app.llm.cache_manager import CacheManager
3 | from app.llm.retry_handler import retry_with_backoff
4 | from app.llm.model_router import ModelRouter
5 | from openai import OpenAI
6 |
7 | settings = get_settings()
8 |
9 | class LLMWrapper:
10 |     def __init__(self, model: str = None):
11 |         self.model = model or settings.model_routing.IDEATION_MODEL
12 |         self.cache = CacheManager()
13 |         self.client = OpenAI(api_key=settings.OPENAI_API_KEY)
14 |
15 |     @retry_with_backoff(retries=settings.llm.MAX_RETRIES, backoff_in_seconds=1)
16 |     def query(self, prompt: str, system_prompt="You are a helpful assistant"):
17 |         cached = self.cache.get(self.model, [{"role": "user", "content": prompt}])
18 |         if cached:
19 |             return cached
20 |
21 |         response = self.client.chat.completions.create(
22 |             model=self.model,
23 |             messages=[
24 |                 {"role": "system", "content": system_prompt},
25 |                 {"role": "user", "content": prompt}
26 |             ],
27 |             temperature=settings.model.TEMPERATURE,
28 |             max_tokens=settings.model.MAX_TOKENS
29 |         )
30 |
31 |         answer = response.choices[0].message.content
32 |         self.cache.set(self.model, [{"role": "user", "content": prompt}], answer)
33 |         return answer
34 |
```

## model\_router.py

```
1 | from app.config.settings import get_settings
2 |
3 | settings = get_settings()
4 |
5 | class ModelRouter:
6 |     MODEL_MAP = {
7 |         "synthesis": settings.model_routing.IDEATION_MODEL,
8 |         "extraction": settings.model_routing.ANALYSIS_MODEL,
9 |         "classification": settings.model_routing.CLASSIFICATION_MODEL
10 |         if hasattr(settings.model_routing, "CLASSIFICATION_MODEL")
11 |         else settings.model_routing.ANALYSIS_MODEL
12 |     }
13 |
14 |     @classmethod
15 |     def get_model_for_task(cls, task_type: str):
16 |         return cls.MODEL_MAP.get(task_type, settings.model_routing.IDEATION_MODEL)
17 |
```



## template.py

```
1 | MARKET_ANALYSIS_TEMPLATE = """
2 | Analyze the following market data for {sector}:
3 | Data: {data}
4 | Focus on: {key_metrics}
5 | """
6 |
7 | def format_analysis_prompt(sector: str, data: str, key_metrics: str) -> str:
8 |     return MARKET_ANALYSIS_TEMPLATE.format(
9 |         sector=sector,
10 |         data=data,
11 |         key_metrics=key_metrics
12 |     )
13 |
```

## retry\_handler.py

```
1 | import time
2 | from functools import wraps
3 |
4 | def retry_with_backoff(retries=3, backoff_in_seconds=1):
5 |     def decorator(func):
6 |         @wraps(func)
7 |         def wrapper(*args, **kwargs):
8 |             attempts = 0
9 |             while attempts < retries:
10 |                 try:
11 |                     return func(*args, **kwargs)
12 |                 except Exception as e:
13 |                     attempts += 1
14 |                     print(f"Error: {e}. Retrying {attempts}/{retries}...")
15 |                     time.sleep(backoff_in_seconds * (2 ** (attempts - 1)))
16 |             return func(*args, **kwargs)
17 |         return wrapper
18 |     return decorator
19 |
20 |
```

1 |  
2 |

## context\_compressor.py

```
1 | from typing import List
2 | from app.rag.schemas import RetrievedDocument
3 |
4 |
5 | class ContextCompressor:
6 |     def __init__(self, max_chars: int = 8000):
7 |         self.max_chars = max_chars
8 |
9 |     def compress(self, docs: List[RetrievedDocument]) -> str:
10 |         context = ""
11 |
12 |         for doc in docs:
13 |             chunk = f"[Source:{doc.metadata.get('source')}] \n{doc.text} \n\n"
14 |
15 |             if len(context) + len(chunk) > self.max_chars:
16 |                 break
17 |
18 |             context += chunk
19 |
20 |         return context.strip()
21 |
```

## insight\_generator.py

```
1 | from app.llm.client import LLMWrapper
2 | from app.llm.model_router import ModelRouter
3 | from app.rag.schemas import InsightResult
4 | import json
5 |
6 |
7 | class InsightGenerator:
8 |     def __init__(self):
9 |         model = ModelRouter.get_model_for_task("synthesis")
10 |         self.llm = LLMWrapper(model=model)
11 |
12 |     def generate(self, query: str, context: str) -> InsightResult:
13 |
14 |         system_prompt = """
15 | You are a senior startup analyst.
16 | Extract:
17 | - key pain points
18 | - market opportunities
19 | - emerging trends
20 | - strong signals
21 | Return JSON.
22 | """
23 |
24 |         prompt = f"""
25 | QUERY:
26 | {query}
27 |
28 | DATA:
29 | {context}
30 |
31 | OUTPUT FORMAT:
32 | {{
33 |     "summary": "",
34 |     "pain_points": [],
35 |     "opportunities": [],
36 |     "signals": []
37 | }}
38 | """
39 |
40 |         response = self.llm.query(prompt, system_prompt=system_prompt)
41 |
42 |         data = json.loads(response)
43 |
44 |         return InsightResult(
45 |             summary=data.get("summary", ""),
46 |             pain_points=data.get("pain_points", []),
47 |             opportunities=data.get("opportunities", []),
48 |             signals=data.get("signals", []),
49 |         )
50 |
```

## painpoint\_clusterer.py

```
1 | from collections import defaultdict
2 | from typing import List
3 |
4 |
5 | class PainPointClusterer:
6 |     def cluster(self, pain_points: List[str]):
7 |
8 |         clusters = defaultdict(list)
9 |
10 |        for p in pain_points:
11 |            key = p.split()[0]
12 |            clusters[key].append(p)
13 |
14 |        return dict(clusters)
15 |
```

## retriever.py

```
1 | from typing import List
2 | from app.vector_db.semantic_search import SemanticSearchEngine
3 | from app.vector_db.config import VectorDBConfig
4 | from app.rag.schemas import RetrievedDocument
5 |
6 |
7 | class RAGRetriever:
8 |     def __init__(self, config: VectorDBConfig):
9 |         self.engine = SemanticSearchEngine(config)
10 |
11 |     def retrieve(self, query: str, top_k: int = 10) -> List[RetrievedDocument]:
12 |         results = self.engine.search(query, top_k=top_k)
13 |
14 |         docs = []
15 |
16 |         ids = results["ids"][0]
17 |         texts = results["documents"][0]
18 |         metas = results["metadatas"][0]
19 |         scores = results["distances"][0]
20 |
21 |         for i in range(len(ids)):
22 |             docs.append(
23 |                 RetrievedDocument(
24 |                     id=ids[i],
25 |                     text=texts[i],
26 |                     metadata=metas[i],
27 |                     score=scores[i],
28 |                 )
29 |             )
30 |
31 |         return docs
32 |
```

## schemas.py

```
1 | from dataclasses import dataclass
2 | from typing import List, Dict, Any
3 |
4 |
5 | @dataclass
6 | class RetrievedDocument:
7 |     id: str
8 |     text: str
9 |     metadata: Dict[str, Any]
10 |     score: float
11 |
12 |
13 | @dataclass
14 | class InsightResult:
15 |     summary: str
16 |     pain_points: List[str]
17 |     opportunities: List[str]
18 |     signals: List[str]
19 |
```



## trend\_detector.py

```
1 | from collections import Counter
2 | from typing import List
3 | from app.rag.schemas import RetrievedDocument
4 |
5 |
6 | class TrendDetector:
7 |     def detect(self, docs: List[RetrievedDocument], top_k=5):
8 |
9 |         words = []
10 |
11 |         for d in docs:
12 |             words.extend(d.text.split())
13 |
14 |         counter = Counter(words)
15 |
16 |         return counter.most_common(top_k)
17 |
```

## cache.py

```
1 | import json
2 | from pathlib import Path
3 |
4 |
5 | class EmbeddingCache:
6 |     def __init__(self, path="data/cache/embeddings.json"):
7 |         self.path = Path(path)
8 |         self.path.parent.mkdir(parents=True, exist_ok=True)
9 |
10 |         if self.path.exists():
11 |             self.cache = json.loads(self.path.read_text())
12 |         else:
13 |             self.cache = {}
14 |
15 |     def get(self, key):
16 |         return self.cache.get(key)
17 |
18 |     def set(self, key, value):
19 |         self.cache[key] = value
20 |         self.path.write_text(json.dumps(self.cache))
21 |
```

## chroma\_client.py

```
1 | import chromadb
2 | from chromadb.config import Settings
3 | from app.vector_db.config import VectorDBConfig
4 |
5 |
6 | class ChromaVectorDB:
7 |     def __init__(self, config: VectorDBConfig):
8 |         self.config = config
9 |         self.client = chromadb.Client(Settings(
10 |             chroma_db_impl="duckdb+parquet",
11 |             persist_directory=self.config.persist_directory,
12 |             anonymized_telemetry=False
13 |         ))
14 |         self.collection = self.client.get_or_create_collection(name=self.config.collection_name)
15 |
16 |     def add_documents(self, ids: list[str], documents: list[str], embeddings: list[list[float]], metadatas: list[dict] | None = None):
17 |         self.collection.add(
18 |             ids=ids,
19 |             documents=documents,
20 |             embeddings=embeddings,
21 |             metadatas=metadatas
22 |         )
23 |
24 |     def query(self, query_embedding, top_k=5):
25 |         return self.collection.query(
26 |             query_embeddings=[query_embedding],
27 |             n_results=top_k
28 |         )
29 |
30 |
```

## config.py

```
1 | from dataclasses import dataclass
2 |
3 | @dataclass
4 | class VectorDBConfig:
5 |     persist_directory: str = "data/vector_db"
6 |     collection_name: str = "startup_ideas"
7 |     embedding_model: str = "text-embedding-3-small"
8 |     batch_size: int = 32
9 |     max_retries: int = 3
10 |
11 |
```

## deduplicator.py

```
1 | import hashlib
2 |
3 |
4 | def hash_text(text: str) -> str:
5 |     return hashlib.sha256(text.encode()).hexdigest()
6 |
```

## embedding\_service.py

```
1 | import time
2 | from openai import OpenAI
3 | from app.vector_db.logger import get_logger
4 | from app.config.settings import get_settings
5 |
6 | settings = get_settings()
7 | logger = get_logger("embedding_service")
8 |
9 |
10 | class EmbeddingService:
11 |     def __init__(self, model: str, max_retries=3):
12 |         self.client = OpenAI(api_key=settings.OPENAI_API_KEY)
13 |         self.model = model
14 |         self.max_retries = max_retries
15 |
16 |     def embed_batch(self, texts):
17 |         for attempt in range(self.max_retries):
18 |             try:
19 |                 response = self.client.embeddings.create(
20 |                     model=self.model,
21 |                     input=texts,
22 |                 )
23 |                 return [e.embedding for e in response.data]
24 |
25 |             except Exception as e:
26 |                 wait = 2 ** attempt
27 |                 logger.warning(f"Embedding failed, retrying in {wait}s | {e}")
28 |                 time.sleep(wait)
29 |
30 |         raise RuntimeError("Embedding failed after retries")
31 |
```

## ingestion\_pipeline.py

```
1 | import json
2 | from tqdm import tqdm
3 |
4 | from app.vector_db.schemas import Document
5 | from app.vector_db.deduplicator import hash_text
6 | from app.vector_db.embedding_service import EmbeddingService
7 | from app.vector_db.cache import EmbeddingCache
8 | from app.vector_db.chroma_client import ChromaVectorDB
9 | from app.vector_db.logger import get_logger
10 | from app.vector_db.config import VectorDBConfig
11 |
12 |
13 | logger = get_logger("ingestion_pipeline")
14 |
15 |
16 | class VectorIngestionPipeline:
17 |     def __init__(self, config: VectorDBConfig):
18 |         self.config = config
19 |         self.db = ChromaVectorDB(config)
20 |         self.embedder = EmbeddingService(
21 |             model=config.embedding_model,
22 |             max_retries=config.max_retries,
23 |         )
24 |         self.cache = EmbeddingCache()
25 |
26 |     def load_json(self, path: str):
27 |         with open(path, "r", encoding="utf-8") as f:
28 |             data = json.load(f)
29 |
30 |         documents = []
31 |
32 |         for item in data:
33 |             text = f"{item.get('title','')} {item.get('body','')}".strip()
34 |
35 |             doc = Document(
36 |                 id=hash_text(text),
37 |                 text=text,
38 |                 metadata={
39 |                     "source": item.get("source"),
40 |                     "score": item.get("score"),
41 |                     "comments": item.get("comments"),
42 |                     "created_at": item.get("created_at"),
43 |                 },
44 |             )
45 |             documents.append(doc)
46 |
47 |         return documents
48 |
49 |     def index(self, documents):
50 |         batch_size = self.config.batch_size
51 |
52 |         for i in tqdm(range(0, len(documents), batch_size)):
53 |             batch = documents[i : i + batch_size]
54 |
55 |             texts = []
56 |             ids = []
57 |             metas = []
58 |             embeddings = []
59 |
60 |             for doc in batch:
61 |                 cached = self.cache.get(doc.id)
62 |
63 |                 if cached:
64 |                     embeddings.append(cached)
65 |                 else:
66 |                     texts.append(doc.text)
67 |
68 |                     ids.append(doc.id)
69 |                     metas.append(doc.metadata)
70 |
71 |             if texts:
72 |                 new_embeddings = self.embedder.embed_batch(texts)
73 |
74 |                 for text, emb in zip(texts, new_embeddings):
75 |                     self.cache.set(hash_text(text), emb)
76 |                     embeddings.append(emb)
77 |
78 |             self.db.add_documents(
79 |                 ids=ids,
80 |                 embeddings=embeddings,
81 |                 documents=[d.text for d in batch],
82 |                 metadatas=metas,
83 |             )
84 |
85 |             logger.info(f"Indexed batch size={len(batch)}")
86 |
```

## logger.py

```
1 | import logging
2 |
3 | def get_logger(name: str):
4 |     logger = logging.getLogger(name)
5 |     logger.setLevel(logging.INFO)
6 |
7 |     if not logger.handlers:
8 |         handler = logging.StreamHandler()
9 |         formatter = logging.Formatter(
10 |             '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
11 |         )
12 |         handler.setFormatter(formatter)
13 |         logger.addHandler(handler)
14 |
15 |     return logger
16 |
```



## schemas.py

```
1 | from dataclasses import dataclass
2 | from typing import Dict, Any
3 |
4 | @dataclass
5 | class Document:
6 |     id: str
7 |     text: str
8 |     metadata: Dict[str, Any]
9 |
```

## semantic\_search.py

```
1 | from openai import OpenAI
2 | from app.vector_db.chroma_client import ChromaVectorDB
3 | from app.vector_db.config import VectorDBConfig
4 | from app.config.settings import get_settings
5 |
6 | settings = get_settings()
7 |
8 |
9 | class SemanticSearchEngine:
10 |     def __init__(self, config: VectorDBConfig):
11 |         self.client = OpenAI(api_key=settings.OPENAI_API_KEY)
12 |         self.model = config.embedding_model
13 |         self.db = ChromaVectorDB(config)
14 |
15 |     def embed_query(self, query):
16 |         response = self.client.embeddings.create(
17 |             model=self.model,
18 |             input=[query],
19 |         )
20 |         return response.data[0].embedding
21 |
22 |     def search(self, query, top_k=5):
23 |         embedding = self.embed_query(query)
24 |         return self.db.query(embedding, top_k)
25 |
```

## similarity.py

```
1 | import math
2 |
3 |
4 | def cosine_similarity(a, b):
5 |     dot = sum(x * y for x, y in zip(a, b))
6 |     na = math.sqrt(sum(x * x for x in a))
7 |     nb = math.sqrt(sum(y * y for y in b))
8 |     return dot / (na * nb) if na and nb else 0.0
9 |
```

## generate\_pdf.py

```
1 | import os
2 | from reportlab.platypus import SimpleDocTemplate, Paragraph, Spacer, Preformatted, PageBreak
3 | from reportlab.lib.styles import getSampleStyleSheet, ParagraphStyle
4 | from reportlab.lib.pagesizes import A4
5 | from reportlab.lib.units import mm
6 | from reportlab.lib import colors
7 | from pygments import highlight
8 | from pygments.lexers import get_lexer_for_filename, TextLexer
9 | from pygments.formatters import HtmlFormatter
10 | from pygments.styles import get_style_by_name
11 | from reportlab.lib.styles import StyleSheet1
12 | from reportlab.platypus import Flowable
13 | from reportlab.lib.enums import TA_CENTER
14 | from reportlab.lib.styles import ParagraphStyle
15 | from reportlab.lib.styles import getSampleStyleSheet
16 | from reportlab.lib.styles import ParagraphStyle
17 | from reportlab.platypus import XPreformatted
18 |
19 |
20 | EXCLUDE_DIRS = {".git", "__pycache__", "venv", ".venv", "node_modules", "build", "dist"}
21 | SUPPORTED_EXTENSIONS = (".py", ".js", ".ts", ".go", ".java", ".cpp", ".c", ".json", ".yaml", ".yml", ".md")
22 |
23 |
24 | def collect_code_files(directory):
25 |     code_files = []
26 |     for root, dirs, files in os.walk(directory):
27 |         dirs[:] = [d for d in dirs if d not in EXCLUDE_DIRS]
28 |
29 |         for file in files:
30 |             if file.endswith(SUPPORTED_EXTENSIONS):
31 |                 code_files.append(os.path.join(root, file))
32 |
33 |     return sorted(code_files)
34 |
35 |
36 | def highlight_code(code, filename):
37 |     try:
38 |         lexer = get_lexer_for_filename(filename)
39 |     except:
40 |         lexer = TextLexer()
41 |
42 |     formatter = HtmlFormatter(style="monokai", noclasses=True)
43 |     return highlight(code, lexer, formatter)
44 |
45 |
46 | def strip_html_tags(text):
47 |     import re
48 |     clean = re.compile("<.*?>")
49 |     return re.sub(clean, "", text)
50 |
51 |
52 | def create_pdf_from_code(files, output_pdf):
53 |     doc = SimpleDocTemplate(
54 |         output_pdf,
55 |         pagesize=A4,
56 |         rightMargin=20,
57 |         leftMargin=20,
58 |         topMargin=20,
59 |         bottomMargin=20,
60 |     )
61 |
62 |     styles = getSampleStyleSheet()
63 |
64 |     title_style = ParagraphStyle(
65 |         name="TitleStyle",
66 |         fontName="Helvetica-Bold",
67 |         fontSize=12,
68 |         spaceAfter=8,
69 |         alignment=TA_CENTER,
70 |     )
71 |
72 |     code_style = ParagraphStyle(
73 |         name="CodeStyle",
74 |         fontName="Courier",
75 |         fontSize=7,
76 |         leading=8,
77 |         backColor=None,
78 |     )
79 |
80 |     elements = []
81 |
82 |     for file in files:
83 |         with open(file, "r", encoding="utf-8", errors="ignore") as f:
84 |             code = f.read()
85 |
86 |             elements.append(Paragraph(os.path.basename(file), title_style))
87 |
88 |             highlighted = highlight_code(code, file)
89 |             clean_code = strip_html_tags(highlighted)
90 |
91 |             numbered_lines = "\n".join(
92 |                 f"{str(i+1).rjust(4)} | {line}"
93 |                 for i, line in enumerate(clean_code.splitlines())
94 |             )
95 |
96 |             elements.append(XPreformatted(numbered_lines, code_style))
```

```
97 |         elements.append(PageBreak())
98 |
99 |     doc.build(elements)
100 |
101 |
102 | if __name__ == "__main__":
103 |     project_dir = "/"
104 |     output_pdf = "project_code_pro.pdf"
105 |
106 |     code_files = collect_code_files(project_dir)
107 |
108 |     print(f"Found {len(code_files)} files")
109 |     create_pdf_from_code(code_files, output_pdf)
110 |
111 |     print(f"PDF created: {output_pdf}")
112 |
```

## data\_pipeline.py

```
1 | import json
2 | import re
3 | from pathlib import Path
4 | from typing import Iterable, List
5 |
6 | from app.config.settings import get_settings
7 | from app.llm.client import LLMWrapper
8 | from app.llm.model_router import ModelRouter
9 | from app.llm.prompts.template import format_analysis_prompt
10 |
11 | settings = get_settings()
12 |
13 | DEFAULT_SECTOR = "startups & emerging tech"
14 |
15 | DEFAULT_KEY_METRICS = (
16 |     "pain points, unmet needs, traction signals, "
17 |     "pricing hints, and competitive alternatives"
18 | )
19 |
20 | def load_json_records(path: Path) -> List[dict]:
21 |     if not path.exists():
22 |         raise FileNotFoundError(f"File not found: {path}")
23 |
24 |     with path.open("r", encoding="utf-8") as handle:
25 |         return json.load(handle)
26 |
27 | def preprocess_text(title: str | None, body: str | None) -> str:
28 |     combined = " ".join([segment for segment in [title, body] if segment]).strip()
29 |
30 |     combined = combined.lower()
31 |     combined = re.sub(r"http\S+|www\.\S+", "", combined)
32 |     combined = re.sub(r"^\w\s\-\.\!?\]", "", combined)
33 |     combined = re.sub(r"\s+", " ", combined)
34 |
35 |     return combined.strip()
36 |
37 | def iter_batches(items: List[str], batch_size: int) -> Iterable[List[str]]:
38 |     for start in range(0, len(items), batch_size):
39 |         yield items[start:start + batch_size]
40 |
41 | def generate_embeddings(texts: List[str]) -> List[List[float]]:
42 |     if not texts:
43 |         return []
44 |
45 |     from openai import OpenAI
46 |
47 |     client = OpenAI(api_key=settings.OPENAI_API_KEY)
48 |
49 |     embeddings: List[List[float]] = []
50 |
51 |     for batch in iter_batches(texts, batch_size=50):
52 |         response = client.embeddings.create(
53 |             model=settings.model.EMBEDDING_MODEL,
54 |             input=batch
55 |         )
56 |
57 |         ordered = sorted(response.data, key=lambda item: item.index)
58 |         embeddings.extend([item.embedding for item in ordered])
59 |
60 |     return embeddings
61 |
62 | def generate_market_analysis(llm: LLMWrapper, text: str, source: str) -> str:
63 |     prompt = format_analysis_prompt(
64 |         sector=DEFAULT_SECTOR,
65 |         data=f"Source: {source}\n\nContent: {text}",
66 |         key_metrics=DEFAULT_KEY_METRICS,
67 |     )
68 |
69 |     return llm.query(
70 |         prompt,
71 |         system_prompt=(
72 |             "You are a senior market research analyst. "
73 |             "Extract startup pain points, unmet needs, and market signals."
74 |         ),
75 |     )
76 |
77 | def build_processed_records(raw_records: List[dict]) -> List[dict]:
78 |
79 |     cleaned_texts = [
80 |         preprocess_text(record.get("title"), record.get("body"))
81 |         for record in raw_records
82 |     ]
83 |
84 |     embeddings = generate_embeddings(cleaned_texts)
85 |
86 |     analysis_model = ModelRouter.get_model_for_task("synthesis")
87 |     llm = LLMWrapper(model=analysis_model)
88 |
89 |     processed_records: List[dict] = []
90 |
91 |     for record, cleaned_text, embedding in zip(
92 |         raw_records, cleaned_texts, embeddings
93 |     ):
94 |         processed_records.append(
95 |             {
96 |                 "title": record.get("title"),
```

```

97 |         "body": record.get("body"),
98 |         "score": record.get("score"),
99 |         "comments": record.get("comments"),
100 |         "created_at": record.get("created_at"),
101 |         "source": record.get("source"),
102 |         "cleaned_text": cleaned_text,
103 |         "embedding": embedding,
104 |         "market_analysis": generate_market_analysis(
105 |             llm,
106 |             cleaned_text,
107 |             record.get("source", "unknown"),
108 |         ),
109 |     }
110 | )
111 |
112 | return processed_records
113 |
114 | def run_pipeline() -> Path:
115 |     raw_path = Path(settings.paths.RAW_DATA)
116 |     processed_path = Path(settings.paths.PROCESSED_DATA)
117 |     processed_path.mkdir(parents=True, exist_ok=True)
118 |
119 |     reddit_records = load_json_records(raw_path / "reddit.json")
120 |     hn_records = load_json_records(raw_path / "hn.json")
121 |
122 |     combined_records = reddit_records + hn_records
123 |
124 |     processed_records = build_processed_records(combined_records)
125 |
126 |     output_file = processed_path / "market_analysis.json"
127 |
128 |     with output_file.open("w", encoding="utf-8") as handle:
129 |         json.dump(processed_records, handle, ensure_ascii=False, indent=2)
130 |
131 |     print(f"Processed {len(processed_records)} records -> {output_file}")
132 |
133 |     return output_file
134 |
135 |
136 | if __name__ == "__main__":
137 |     run_pipeline()
138 |

```

1 |  
2 |



1 |  
2 |

## generate\_report.py

```
1  """
2  Script to generate market intelligence reports
3
4  This script can generate reports in multiple ways:
5  1. From processed data (market_analysis.json)
6  2. Using the multi-agent system for comprehensive analysis
7  3. Using the RAG insight generator
8
9  Usage:
10     python scripts/generate_report.py [--method METHOD] [--topic TOPIC] [--output OUTPUT]
11
12  Options:
13     --method: 'data' (from processed data), 'agents' (multi-agent), 'rag' (RAG insights)
14     --topic: Topic for agent-based or RAG-based reports
15     --output: Output file path (default: reports/report_YYYYMMDD_HHMMSS.json)
16  """
17
18  import os
19  import sys
20  import json
21  import argparse
22  from pathlib import Path
23  from datetime import datetime
24
25  # Add project root to path
26  project_root = Path(__file__).parent.parent
27  sys.path.insert(0, str(project_root))
28
29  from app.config.settings import get_settings
30  from app.rag.insight_generator import InsightGenerator
31  from app.rag.retriever import RAGRetriever
32  from app.vector_db.config import VectorDBConfig
33  from app.agents import Orchestrator
34
35  settings = get_settings()
36
37  def generate_report_from_data(output_path: Path) -> dict:
38      """Generate report from processed data"""
39      processed_file = Path(settings.paths.PROCESSED_DATA) / "market_analysis.json"
40
41      if not processed_file.exists():
42          raise FileNotFoundError(f"Processed data not found: {processed_file}")
43
44      with open(processed_file, "r", encoding="utf-8") as f:
45          data = json.load(f)
46
47      # Aggregate insights
48      all_analyses = [record.get("market_analysis", "") for record in data]
49
50      # Count sources
51      sources = {}
52      for record in data:
53          source = record.get("source", "unknown")
54          sources[source] = sources.get(source, 0) + 1
55
56      # Generate summary using LLM
57      from app.llm.client import LLMWrapper
58      from app.llm.model_router import ModelRouter
59
60      model = ModelRouter.get_model_for_task("synthesis")
61      llm = LLMWrapper(model=model)
62
63      summary_prompt = f"""Based on {len(data)} market analysis records, generate a comprehensive market intelligence report.
64
65  The report should include:
66  1. Executive Summary
67  2. Key Market Trends
68  3. Pain Points Identified
69  4. Opportunities
70  5. Recommendations
71
72  Analysis excerpts:
73  {chr(10).join(all_analyses[:10])}
74
75  Generate a structured report in JSON format:
76  {{
77  {{
78      "executive_summary": "...",
79      "key_trends": ["...", "..."],
80      "pain_points": ["...", "..."],
81      "opportunities": ["...", "..."],
82      "recommendations": ["...", "..."]
83  }}
84  }}"""
85
86      response = llm.query(
87          summary_prompt,
88          system_prompt="You are a senior market intelligence analyst creating executive reports."
89      )
90
91      try:
92          report_data = json.loads(response)
93      except json.JSONDecodeError:
94          report_data = {
95              "executive_summary": response[:500],
96              "key_trends": [],
97              "pain_points": [],
```

```

97         "opportunities": [],
98         "recommendations": []
99     }
100
101     # Add metadata
102     report = {
103         "report_type": "data_analysis",
104         "generated_at": datetime.now().isoformat(),
105         "data_sources": sources,
106         "total_records": len(data),
107         "report": report_data
108     }
109
110     return report
111
112
113 def generate_report_from_agents(topic: str, output_path: Path) -> dict:
114     """Generate report using multi-agent system"""
115     print(f"■ Using multi-agent system for topic: {topic}")
116
117     vector_config = VectorDBConfig(
118         persist_directory=settings.vectordb.PERSIST_DIR,
119         collection_name=settings.vectordb.COLLECTION_NAME,
120         embedding_model=settings.model.EMBEDDING_MODEL
121     )
122
123     orchestrator = Orchestrator(vector_db_config=vector_config)
124
125     result = orchestrator.run_full_pipeline(
126         topic=topic,
127         generate_ideas=True,
128         evaluate_ideas=True,
129         num_ideas=5,
130         top_k_research=10
131     )
132
133     # Convert to report format
134     report = {
135         "report_type": "multi_agent",
136         "generated_at": datetime.now().isoformat(),
137         "topic": topic,
138         "execution_time": result.execution_time,
139         "research": {
140             "summary": result.research_results.summary if result.research_results else "",
141             "key_findings": result.research_results.key_findings if result.research_results else [],
142             "sources": result.research_results.sources if result.research_results else []
143         } if result.research_results else None,
144         "market_analysis": {
145             "market_size": result.market_analysis.market_size if result.market_analysis else None,
146             "market_maturity": result.market_analysis.market_maturity if result.market_analysis else None,
147             "trends": result.market_analysis.trends if result.market_analysis else [],
148             "opportunities": result.market_analysis.opportunities if result.market_analysis else [],
149             "threats": result.market_analysis.threats if result.market_analysis else []
150         } if result.market_analysis else None,
151         "generated_ideas": [
152             {
153                 "title": idea.title,
154                 "description": idea.description,
155                 "target_audience": idea.target_audience,
156                 "value_proposition": idea.value_proposition,
157                 "key_features": idea.key_features,
158                 "market_opportunity": idea.market_opportunity
159             }
160             for idea in (result.generated_ideas or [])
161         ],
162         "evaluations": [
163             {
164                 "idea": eval_result.idea[:100] + "..." if len(eval_result.idea) > 100 else eval_result.idea,
165                 "overall_score": eval_result.overall_score,
166                 "feasibility_score": eval_result.feasibility_score,
167                 "market_potential_score": eval_result.market_potential_score,
168                 "innovation_score": eval_result.innovation_score,
169                 "verdict": eval_result.verdict,
170                 "strengths": eval_result.strengths,
171                 "weaknesses": eval_result.weaknesses,
172                 "recommendations": eval_result.recommendations
173             }
174             for eval_result in (result.evaluations or [])
175         ],
176         "top_ideas": [
177             {
178                 "title": idea.title,
179                 "description": idea.description,
180                 "value_proposition": idea.value_proposition
181             }
182             for idea in (result.top_ideas or [])
183         ],
184         "summary": result.summary
185     }
186
187     return report
188
189
190 def generate_report_from_rag(topic: str, output_path: Path) -> dict:
191     """Generate report using RAG insight generator"""
192     print(f"■ Using RAG system for topic: {topic}")
193
194     vector_config = VectorDBConfig(

```

```

195         persist_directory=settings.vectordb.PERSIST_DIR,
196         collection_name=settings.vectordb.COLLECTION_NAME,
197         embedding_model=settings.model.EMBEDDING_MODEL
198     )
199
200     retriever = RAGRetriever(vector_config)
201     insight_generator = InsightGenerator()
202
203     # Retrieve relevant documents
204     documents = retriever.retrieve(topic, top_k=10)
205
206     # Format context
207     context = "\n\n".join([
208         f"Document {i+1} (Score: {doc.score:.3f}): \n{doc.text[:500]}..."
209         for i, doc in enumerate(documents)
210     ])
211
212     # Generate insights
213     insights = insight_generator.generate(topic, context)
214
215     # Create report
216     report = {
217         "report_type": "rag_insights",
218         "generated_at": datetime.now().isoformat(),
219         "topic": topic,
220         "documents_analyzed": len(documents),
221         "insights": {
222             "summary": insights.summary,
223             "pain_points": insights.pain_points,
224             "opportunities": insights.opportunities,
225             "signals": insights.signals
226         },
227         "sources": list(set([doc.metadata.get("source", "unknown") for doc in documents]))
228     }
229
230     return report
231
232
233 def save_report(report: dict, output_path: Path):
234     """Save report to file"""
235     output_path.parent.mkdir(parents=True, exist_ok=True)
236
237     with open(output_path, "w", encoding="utf-8") as f:
238         json.dump(report, f, ensure_ascii=False, indent=2)
239
240     print(f"\n■ Report saved to: {output_path}")
241
242
243 def print_report_summary(report: dict):
244     """Print a summary of the report"""
245     print("\n" + "=" * 60)
246     print("Report Summary")
247     print("=" * 60)
248
249     report_type = report.get("report_type", "unknown")
250     print(f"\n■ Report Type: {report_type}")
251     print(f"■ Generated: {report.get('generated_at', 'Unknown')}")
252
253     if report_type == "data_analysis":
254         print(f"■ Records Analyzed: {report.get('total_records', 0)}")
255         report_data = report.get("report", {})
256         print(f"■ Trends Identified: {len(report_data.get('key_trends', []))}")
257         print(f"■ Opportunities: {len(report_data.get('opportunities', []))}")
258
259     elif report_type == "multi_agent":
260         print(f"■ Execution Time: {report.get('execution_time', 0):.2f}s")
261         if report.get("research"):
262             print(f"■ Research Findings: {len(report['research'].get('key_findings', []))}")
263         if report.get("market_analysis"):
264             print(f"■ Market Maturity: {report['market_analysis'].get('market_maturity', 'Unknown')}")
265         print(f"■ Ideas Generated: {len(report.get('generated_ideas', []))}")
266         print(f"■ Top Ideas: {len(report.get('top_ideas', []))}")
267
268     elif report_type == "rag_insights":
269         print(f"■ Documents Analyzed: {report.get('documents_analyzed', 0)}")
270         insights = report.get("insights", {})
271         print(f"■ Pain Points: {len(insights.get('pain_points', []))}")
272         print(f"■ Opportunities: {len(insights.get('opportunities', []))}")
273         print(f"■ Signals: {len(insights.get('signals', []))}")
274
275
276 def main():
277     """Main function"""
278     parser = argparse.ArgumentParser(description="Generate market intelligence reports")
279     parser.add_argument(
280         "--method",
281         choices=["data", "agents", "rag"],
282         default="data",
283         help="Report generation method (default: data)"
284     )
285     parser.add_argument(
286         "--topic",
287         type=str,
288         help="Topic for agent-based or RAG-based reports (required for agents/rag methods)"
289     )
290     parser.add_argument(
291         "--output",
292         type=str,

```

```

293 |         help="Output file path (default: reports/report_YYYYMMDD_HHMMSS.json)"
294 |     )
295 |
296 |     args = parser.parse_args()
297 |
298 |     # Validate arguments
299 |     if args.method in ["agents", "rag"] and not args.topic:
300 |         print(f"■ Error: --topic is required for method '{args.method}'")
301 |         return 1
302 |
303 |     # Determine output path
304 |     if args.output:
305 |         output_path = Path(args.output)
306 |     else:
307 |         timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
308 |         reports_dir = Path(settings.paths.REPORTS)
309 |         output_path = reports_dir / f"report_{timestamp}.json"
310 |
311 |     print("=" * 60)
312 |     print("Market Intelligence Report Generator")
313 |     print("=" * 60)
314 |
315 |     try:
316 |         # Generate report based on method
317 |         if args.method == "data":
318 |             print("\n■ Generating report from processed data...")
319 |             report = generate_report_from_data(output_path)
320 |
321 |             elif args.method == "agents":
322 |                 report = generate_report_from_agents(args.topic, output_path)
323 |
324 |             elif args.method == "rag":
325 |                 report = generate_report_from_rag(args.topic, output_path)
326 |
327 |         # Save report
328 |         save_report(report, output_path)
329 |
330 |         # Print summary
331 |         print_report_summary(report)
332 |
333 |         return 0
334 |
335 |     except FileNotFoundError as e:
336 |         print(f"\n■ Error: {str(e)}")
337 |         if args.method == "data":
338 |             print("    Run 'python scripts/run_analysis.py' first to process data.")
339 |         return 1
340 |     except Exception as e:
341 |         print(f"\n■ Error generating report: {str(e)}")
342 |         import traceback
343 |         traceback.print_exc()
344 |         return 1
345 |
346 |
347 | if __name__ == "__main__":
348 |     exit(main())
349 |

```

## run\_agents.py

```
1  """
2  Example script demonstrating the Multi-Agent Market Intelligence System
3
4  Usage:
5      python scripts/run_agents.py
6  """
7
8  from app.agents import Orchestrator
9  from app.vector_db.config import VectorDBConfig
10 from app.config.settings import get_settings
11
12 settings = get_settings()
13
14
15 def main():
16     """Run the multi-agent system"""
17
18     # Initialize vector DB config
19     vector_config = VectorDBConfig(
20         persist_directory=settings.vectordb.PERSIST_DIR,
21         collection_name=settings.vectordb.COLLECTION_NAME,
22         embedding_model=settings.model.EMBEDDING_MODEL
23     )
24
25     # Create orchestrator
26     orchestrator = Orchestrator(vector_db_config=vector_config)
27
28     # Example: Research and generate ideas for a topic
29     topic = "AI-powered productivity tools for remote teams"
30
31     print("=" * 60)
32     print("Multi-Agent Market Intelligence System")
33     print("=" * 60)
34     print(f"\nTopic: {topic}\n")
35
36     # Run full pipeline
37     result = orchestrator.run_full_pipeline(
38         topic=topic,
39         generate_ideas=True,
40         evaluate_ideas=True,
41         num_ideas=5,
42         top_k_research=10
43     )
44
45     # Display results
46     print("\n" + "=" * 60)
47     print("RESULTS SUMMARY")
48     print("=" * 60)
49
50     if result.research_results:
51         print(f"\n■ Research Results:")
52         print(f"    Documents found: {len(result.research_results.documents)}")
53         print(f"    Key findings: {len(result.research_results.key_findings)}")
54         print(f"    Sources: {' , '.join(result.research_results.sources[:3])}...")
55
56     if result.market_analysis:
57         print(f"\n■ Market Analysis:")
58         print(f"    Market Maturity: {result.market_analysis.market_maturity}")
59         print(f"    Market Size: {result.market_analysis.market_size}")
60         print(f"    Opportunities: {len(result.market_analysis.opportunities)}")
61         print(f"    Trends: {' , '.join(result.market_analysis.trends[:3])}...")
62
63     if result.generated_ideas:
64         print(f"\n■ Generated Ideas ({len(result.generated_ideas)}):")
65         for i, idea in enumerate(result.generated_ideas, 1):
66             print(f"    {i}. {idea.title}")
67             print(f"        {idea.description[:100]}...")
68             print(f"        Target: {idea.target_audience}")
69
70     if result.evaluations:
71         print(f"\n■ Evaluations:")
72         for i, eval_result in enumerate(result.evaluations, 1):
73             print(f"    Idea {i}:")
74             print(f"        Overall Score: {eval_result.overall_score:.2f}/1.0")
75             print(f"        Verdict: {eval_result.verdict}")
76             print(f"        Strengths: {len(eval_result.strengths)}")
77
78     if result.top_ideas:
79         print(f"\n■ Top {len(result.top_ideas)} Ideas:")
80         for i, idea in enumerate(result.top_ideas, 1):
81             eval_result = result.evaluations[i-1] if i <= len(result.evaluations) else None
82             score = eval_result.overall_score if eval_result else 0.0
83             print(f"    {i}. {idea.title} (Score: {score:.2f})")
84             print(f"        {idea.value_proposition}")
85
86     print(f"\n■ Execution Time: {result.execution_time:.2f}s")
87     print(f"■ Summary: {result.summary}")
88     print("\n" + "=" * 60)
89
90
91 if __name__ == "__main__":
92     main()
93
```

## run\_analysis.py

```
1  """
2  Script to run data analysis pipeline
3
4  This script:
5  1. Loads raw scraped data (Reddit and Hacker News)
6  2. Preprocesses and cleans the text
7  3. Generates embeddings
8  4. Runs LLM analysis to extract market insights
9  5. Saves processed data to data/processed/
10
11 Usage:
12     python scripts/run_analysis.py
13 """
14
15 import os
16 import sys
17 from pathlib import Path
18
19 # Add project root to path
20 project_root = Path(__file__).parent.parent
21 sys.path.insert(0, str(project_root))
22
23 from pipelines.data_pipeline import run_pipeline, load_json_records
24 from app.config.settings import get_settings
25
26 settings = get_settings()
27
28
29 def main():
30     """Run the data analysis pipeline"""
31     print("=" * 60)
32     print("Data Analysis Pipeline")
33     print("=" * 60)
34
35     raw_path = Path(settings.paths.RAW_DATA)
36     processed_path = Path(settings.paths.PROCESSED_DATA)
37
38     # Check if raw data exists
39     reddit_file = raw_path / "reddit.json"
40     hn_file = raw_path / "hn.json"
41
42     files_exist = []
43     if reddit_file.exists():
44         files_exist.append(("Reddit", reddit_file))
45     if hn_file.exists():
46         files_exist.append(("Hacker News", hn_file))
47
48     if not files_exist:
49         print(f"\n■ No raw data files found in {raw_path}")
50         print("    Please run 'python scripts/run_scrapers.py' first to collect data.")
51         return 1
52
53     print(f"\n■ Found {len(files_exist)} data source(s):")
54     for name, file_path in files_exist:
55         try:
56             records = load_json_records(file_path)
57             print(f"    ■ {name}: {len(records)} records")
58         except Exception as e:
59             print(f"    ■ {name}: Error loading - {str(e)}")
60         return 1
61
62     # Check for OpenAI API key
63     if not settings.OPENAI_API_KEY:
64         print("\n■ OPENAI_API_KEY not found in settings")
65         print("    Please set it in your .env file or environment variables.")
66         return 1
67
68     print(f"\n■ Configuration:")
69     print(f"    Embedding Model: {settings.model.EMBEDDING_MODEL}")
70     print(f"    Analysis Model: {settings.model_routing.ANALYSIS_MODEL}")
71     print(f"    Temperature: {settings.model.TEMPERATURE}")
72     print(f"    Max Tokens: {settings.model.MAX_TOKENS}")
73
74     # Run pipeline
75     try:
76         print("\n■ Starting analysis pipeline...")
77         print("    This may take a while depending on the amount of data...")
78
79         output_file = run_pipeline()
80
81         print("\n" + "=" * 60)
82         print("Pipeline Completed Successfully!")
83         print("=" * 60)
84         print(f"\n■ Processed data saved to: {output_file}")
85
86         # Show statistics
87         try:
88             import json
89             with open(output_file, "r", encoding="utf-8") as f:
90                 processed_data = json.load(f)
91
92             print(f"\n■ Statistics:")
93             print(f"    Total records processed: {len(processed_data)}")
94
95             # Count by source
96             sources = {}
```

```

97 |         for record in processed_data:
98 |             source = record.get("source", "unknown")
99 |             sources[source] = sources.get(source, 0) + 1
100 |
101 |         print(f"    Sources:")
102 |         for source, count in sources.items():
103 |             print(f"        - {source}: {count} records")
104 |
105 |         print(f"\n■ Next steps:")
106 |         print(f"    1. Index data into vector DB (if not already done)")
107 |         print(f"    2. Run 'python scripts/generate_report.py' to generate insights")
108 |         print(f"    3. Or use the multi-agent system: 'python scripts/run_agents.py'")
109 |
110 |     except Exception as e:
111 |         print(f"    ■■ Could not load statistics: {str(e)}")
112 |
113 |     return 0
114 |
115 | except FileNotFoundError as e:
116 |     print(f"\n■ File not found: {str(e)}")
117 |     print("    Make sure raw data files exist in data/raw/")
118 |     return 1
119 | except Exception as e:
120 |     print(f"\n■ Pipeline failed: {str(e)}")
121 |     import traceback
122 |     traceback.print_exc()
123 |     return 1
124 |
125 |
126 | if __name__ == "__main__":
127 |     exit(main())
128 |

```



## run\_scrapers.py

```
1  """
2  Script to run data collection scrapers (Reddit and Hacker News)
3
4  Usage:
5      python scripts/run_scrapers.py
6  """
7
8  import os
9  import sys
10 from pathlib import Path
11
12 # Add project root to path
13 project_root = Path(__file__).parent.parent
14 sys.path.insert(0, str(project_root))
15
16 from app.data_collectors.reddit_scraper import scrape_reddit
17 from app.data_collectors.hn_scraper import scrape_hackernews
18 from app.config.settings import get_settings
19
20 settings = get_settings()
21
22
23 def main():
24     """Run all scrapers"""
25     print("=" * 60)
26     print("Data Collection Scrapers")
27     print("=" * 60)
28
29     # Check for required environment variables for Reddit
30     reddit_required = ["REDDIT_CLIENT_ID", "REDDIT_CLIENT_SECRET", "REDDIT_USER_AGENT"]
31     reddit_missing = [var for var in reddit_required if not os.getenv(var)]
32
33     if reddit_missing:
34         print(f"\n■■ Warning: Missing Reddit environment variables: {' '.join(reddit_missing)}")
35         print("  Reddit scraping will be skipped.")
36         print("  Set these in your .env file or environment.")
37         reddit_enabled = False
38     else:
39         reddit_enabled = True
40
41     results = {}
42
43     # Scrape Reddit
44     if reddit_enabled:
45         try:
46             print("\n■ Scraping Reddit...")
47             reddit_path = scrape_reddit(limit=settings.scrapers.REDDIT_LIMIT)
48             results["reddit"] = {
49                 "status": "success",
50                 "path": reddit_path
51             }
52             print(f"■ Reddit scraping completed")
53         except Exception as e:
54             print(f"■ Reddit scraping failed: {str(e)}")
55             results["reddit"] = {
56                 "status": "failed",
57                 "error": str(e)
58             }
59     else:
60         results["reddit"] = {
61             "status": "skipped",
62             "reason": "Missing environment variables"
63         }
64
65     # Scrape Hacker News
66     try:
67         print("\n■ Scraping Hacker News...")
68         hn_path = scrape_hackernews(limit=settings.scrapers.HN_LIMIT)
69         results["hackernews"] = {
70             "status": "success",
71             "path": hn_path
72         }
73         print(f"■ Hacker News scraping completed")
74     except Exception as e:
75         print(f"■ Hacker News scraping failed: {str(e)}")
76         results["hackernews"] = {
77             "status": "failed",
78             "error": str(e)
79         }
80
81     # Summary
82     print("\n" + "=" * 60)
83     print("Scraping Summary")
84     print("=" * 60)
85
86     for source, result in results.items():
87         status_icon = "■" if result["status"] == "success" else "■" if result["status"] == "failed" else "■■"
88         print(f"{status_icon} {source.capitalize()}: {result['status']}")
89         if result["status"] == "success":
90             print(f"  Path: {result['path']}")
91         elif result["status"] == "failed":
92             print(f"  Error: {result.get('error', 'Unknown error')}")
93
94     # Check if we have data for analysis
95     success_count = sum(1 for r in results.values() if r["status"] == "success")
96     if success_count > 0:
```

```
97 |         print(f"\n■ Successfully scraped {success_count} source(s)")
98 |         print("    Next step: Run 'python scripts/run_analysis.py' to process the data")
99 |     else:
100 |         print("\n■■ No data was successfully scraped. Please check errors above.")
101 |         return 1
102 |
103 |     return 0
104 |
105 |
106 | if __name__ == "__main__":
107 |     exit(main())
108 |
```