Member-only story

Building Anthropic's Multi Agent Research System to Outperform Claude Opus 4 By 90%

34 min read · Jun 21, 2025

Agent Native Follow

The following is end-to-end implementation of the blueprint that Anthropic shared for building multi-agent research system, which significantly outperformed single agent workflows of Claude Opus 4.

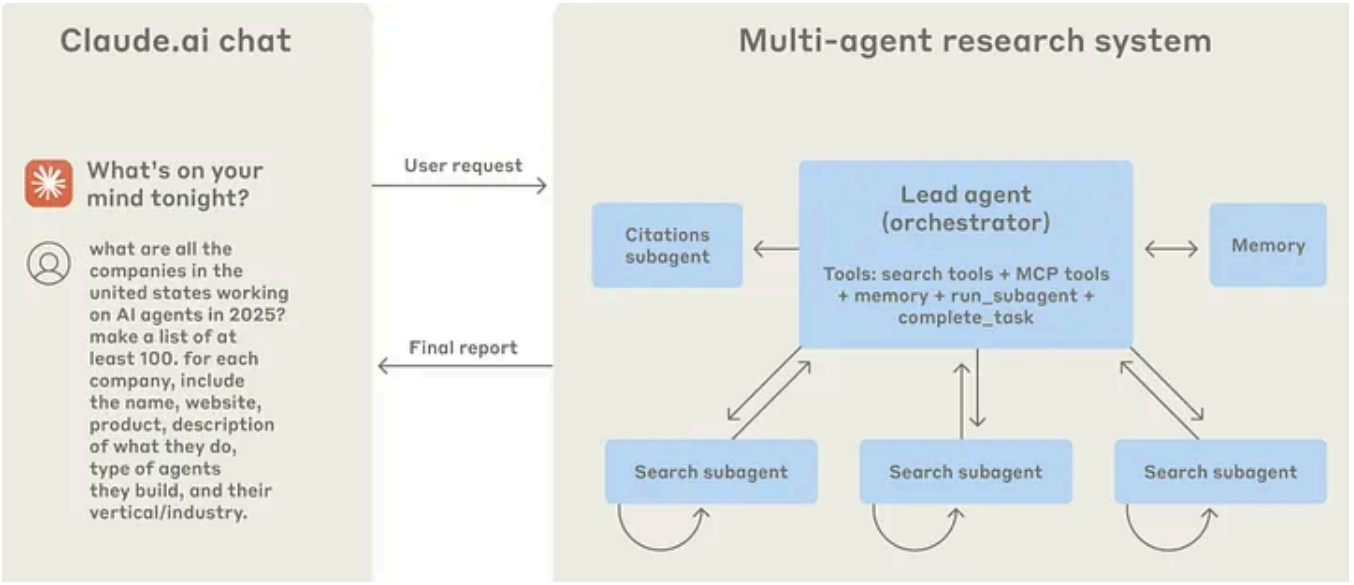
As we have built more advanced version of this system in-house, we wanted to share the simplified blueprint that you can use as a starting point.

Don't forget to check out "Closing Thoughts on Implementation" at the end for lessons learned from building advanced version.

Here's what Anthropic shared in their article, which was a claim to go after.

> *We found that a multi-agent system with Claude Opus 4 as the lead agent and Claude Sonnet 4 subagents outperformed single-agent Claude Opus 4 by 90.2% on our internal research eval.*

And here's a high-level overview of that multi-agent research system:



The multi-agent architecture in action: user queries flow through a lead agent that creates specialized subagents to search for different aspects in parallel.

We'll build the simplied version of this system based on the following components:

- Lead Agent that orchestrates research
- Multiple Search Subagents working in parallel
- Memory persistence
- Basic citation tracking
- A REST API built with FastAPI
- Proper error handling and state management

Let's dive into it.

Anthropic Hackathon

Step 1: Project Setup and Dependencies

First, let's set up our project structure:

```
mkdir multi-agent-researcher
cd multi-agent-researcher
python3 -m venv researcher-venv
source researcher-venv/bin/activate
```

Then create a requirements.txt file

```
fastapi
uvicorn
pydantic
httpx
beautifulsoup4
litellm
```

```
anthropic
python-dotenv
redis
asyncio
```

Here, each dependency reflects the key requirements of a multi-agent system, without bloating the project.

Mainly, `fastapi` for async support, `anthropic` for claude models (`litellm` for model provide flexibility, `httpx` for async HTTP requests for web searching, `beautifulsoup4` for parsing HTML content from search results and `redis` for our distributed memory store, allowing agents to share context and survive restarts.

Each dependency was selected for its async capabilities and production readiness.

Let's install dependencies:

```
pip3 install -r requirements.txt
```

Step 2: Project Structure

Let's also create the following structure, you can get the starter repo here.

```
multi-agent-research/
├── app/
│   ├── __init__.py
│   ├── main.py
│   ├── models/
│   │   ├── __init__.py
│   │   └── schemas.py
│   ├── agents/
│   │   ├── __init__.py
│   │   ├── base_agent.py
│   │   ├── lead_agent.py
│   │   ├── search_agent.py
│   │   └── citation_agent.py
│   ├── tools/
│   │   ├── __init__.py
│   │   ├── search_tools.py
│   │   └── memory_tools.py
```

```
│   ├── core/
│   │   ├── __init__.py
│   │   ├── config.py
│   │   └── prompts.py
│   └── services/
│       ├── __init__.py
│       └── research_service.py
├── .env
├── .gitignore
├── requirements.txt
└── README.md
```

The structure follows a modular architecture pattern that separates concerns clearly.

The `agents/` directory contains all agent logic, with a base class establishing common behavior and specialized agents inheriting from it. The `tools/` directory houses reusable tools that agents can invoke , this separation allows tools to be shared across different agent types. The `services/` layer provides a clean interface between the API endpoints and the agent system, handling orchestration and state management.

This structure makes it easy to add new agent types or tools without modifying existing code.

Step 3: Core Models and Configuration

Let's start with our data models:

`app/models/schemas.py`:

```python
from pydantic import BaseModel, Field
from typing import List, Optional, Dict, Any, Literal
from datetime import datetime
from uuid import UUID, uuid4

class ResearchQuery(BaseModel):
    """User's research query"""
    query: str
    max_subagents: int = Field(default=3, ge=1, le=10)
    max_iterations: int = Field(default=5, ge=1, le=20)
```

```python
class SubAgentTask(BaseModel):
    """Task definition for a subagent"""
    task_id: UUID = Field(default_factory=uuid4)
    objective: str
    search_focus: str
    expected_output_format: str
    max_searches: int = Field(default=5)
    status: Literal["pending", "running", "completed", "failed"] = "pending"

class SearchResult(BaseModel):
    """Result from a search operation"""
    url: str
    title: str
    snippet: str
    content: Optional[str] = None
    relevance_score: float = Field(ge=0.0, le=1.0)

class SubAgentResult(BaseModel):
    """Result from a subagent's research"""
    task_id: UUID
    findings: List[Dict[str, Any]]
    sources: List[SearchResult]
    summary: str
    token_count: int

class ResearchPlan(BaseModel):
    """Research plan created by lead agent"""
    plan_id: UUID = Field(default_factory=uuid4)
    strategy: str
    subtasks: List[SubAgentTask]
    estimated_complexity: Literal["simple", "moderate", "complex"]

class ResearchResult(BaseModel):
    """Final research result"""
    research_id: UUID
    query: str
    report: str
    citations: List[Dict[str, Any]]
    sources_used: List[SearchResult]
    total_tokens_used: int
    execution_time: float
    created_at: datetime = Field(default_factory=datetime.utcnow)
```

Here, Pydantic models serve as both data validation and documentation.

Each model represents a key concept in the multi-agent system.

`ResearchQuery` validates user input and sets sensible defaults to prevent runaway agent behavior. `SubAgentTask` encapsulates everything a subagent needs to work independently, including a unique ID for tracking. The status field enables monitoring of distributed tasks.

`SearchResult` standardizes how we handle information from various sources, with relevance scoring built in for quality filtering.

The configuration class centralizes all settings, making it easy to adjust behavior for different environments without code changes.

`app/core/config.py`:

```
import os
from dotenv import load_dotenv

load_dotenv()

class Settings:
    # API Keys
    ANTHROPIC_API_KEY: str = os.getenv("ANTHROPIC_API_KEY", "")

    # Model Configuration
    LEAD_AGENT_MODEL: str = "claude-3-opus-20240229"
    SUBAGENT_MODEL: str = "claude-3-sonnet-20240229"
    CITATION_MODEL: str = "claude-3-haiku-20240307"

    # Agent Configuration
    MAX_THINKING_LENGTH: int = 50000
    MAX_CONTEXT_LENGTH: int = 200000
    MAX_PARALLEL_SUBAGENTS: int = 5

    # Tool Configuration
    SEARCH_TIMEOUT: int = 30
    MAX_SEARCH_RESULTS: int = 10

    # Memory Configuration
    REDIS_URL: str = os.getenv("REDIS_URL", "redis://localhost:6379")
    MEMORY_TTL: int = 3600  # 1 hour

    # Rate Limiting
    MAX_TOKENS_PER_REQUEST: int = 100000

settings = Settings()
```

Before diving into agent implementations, please review this process diagram showing the complete workflow between different components and agents

With that in mind, we can start the core parts of the implementation.

Step 4: Base Agent Implementation

Let's quickly skim through code, and we will provide explanations at the end of this section.

`app/agents/base_agent.py`:

```
from abc import ABC, abstractmethod
from typing import Dict, Any, List, Optional
```

```python
import asyncio
import anthropic
from app.core.config import settings
import json

class BaseAgent(ABC):
    """Base class for all agents in the system"""

    def __init__(self, model: str, name: str):
        self.model = model
        self.name = name
        self.client = anthropic.AsyncAnthropic(api_key=settings.ANTHROPIC_API_K
        self.conversation_history: List[Dict[str, str]] = []
        self.total_tokens = 0

    @abstractmethod
    def get_system_prompt(self) -> str:
        """Return the system prompt for this agent"""
        pass

    async def think(self, context: str) -> Dict[str, Any]:
        """
        Use extended thinking mode to plan approach
        Returns structured thinking output
        """
        thinking_prompt = f"""
        <thinking>
        Context: {context}

        Please analyze this situation and plan your approach. Consider:
        1. What is the main objective?
        2. What tools or resources do I need?
        3. What steps should I take?
        4. What potential challenges might I face?

        Output your thinking as a JSON object with keys:
        - objective: string
        - approach: string
        - steps: list of strings
        - challenges: list of strings
        </thinking>
        """

        response = await self._call_llm(thinking_prompt, max_tokens=2000)

        # Parse thinking output
        try:
            thinking_text = response.split("<thinking>")[1].split("</thinking>"
            # Simple JSON extraction (in production, use proper parsing)
            return json.loads(thinking_text)
        except:
            return {
                "objective": context,
```

```
                "approach": "Direct investigation",
                "steps": ["Analyze query", "Search for information", "Synthesiz
                "challenges": ["Unknown"]
            }

    async def _call_llm(self, prompt: str, max_tokens: int = 4000) -> str:
        """Make a call to the LLM"""
        try:
            message = await self.client.messages.create(
                model=self.model,
                max_tokens=max_tokens,
                temperature=0.7,
                system=self.get_system_prompt(),
                messages=self.conversation_history + [{"role": "user", "content
            )

            response = message.content[0].text
            self.total_tokens += message.usage.total_tokens

            # Update conversation history
            self.conversation_history.append({"role": "user", "content": prompt
            self.conversation_history.append({"role": "assistant", "content": r

            return response

        except Exception as e:
            print(f"Error calling LLM: {e}")
            raise

    def reset_conversation(self):
        """Reset the conversation history"""
        self.conversation_history = []
        self.total_tokens = 0
```

The base agent establishes patterns that all agents follow.

The conversation history tracking allows agents to maintain context across multiple LLM calls, essential for complex reasoning tasks.

The `think` method implements the “thinking before acting” pattern mentioned in the Anthropic’s blueprint, agents plan their approach before executing, leading to more coherent behavior.

Token counting is built into every LLM call, providing cost visibility. The error handling in `_call_llm` ensures that transient API failures don’t crash the entire research process.

The reset capability allows agents to be reused for multiple tasks, important for resource efficiency.

> *We publish "how-to" guides and thought pieces for startups and solo founders!*
>
> *We pour our* ***passion, expertise, and countless hours into creating content*** *that we believe can make a difference in your journey.*
>
> *But only 1% of our readers follow or engage with us on Medium.*
>
> ***If you ever found value in our content,*** *it would mean a lot if you could* ***follow Agent Native on Medium****, give this article a clap, and drop a hello in the comments!*
>
> ***It's a small gesture but it tremendously helps us deliver much better content and guides for you!***
>
> *Thank you for taking your time to be here, we* really appreciate it.

Step 5: Prompts Configuration

These prompts encode the behavioral patterns that make the system work effectively.

The lead agent prompt emphasizes coordination and delegation skills, teaching it to break down complex queries appropriately.

The scaling rules (1 agent for simple queries, 3–5 for complex ones) prevent resource waste while ensuring thorough research.

The search subagent prompt implements the "start wide, then narrow" strategy from the blueprint, beginning with broad searches before drilling down.

The quality criteria help agents distinguish authoritative sources from SEO spam.

The citation agent prompt ensures academic-style attribution, crucial for research credibility.

`app/core/prompts.py` :

```
LEAD_AGENT_PROMPT = """You are a Lead Research Agent responsible for coordinati

Your responsibilities:
1. Analyze the user's query and determine its complexity
2. Develop a research strategy and break it down into subtasks
3. Create and delegate tasks to specialized search subagents
4. Synthesize results from multiple subagents into a coherent report
5. Ensure all claims are properly supported by sources

When creating subtasks for subagents:
- Be specific about what information they should find
- Avoid overlapping responsibilities between agents
- Scale the number of agents to match query complexity:
  - Simple fact-finding: 1 agent with 3-5 searches
  - Comparisons: 2-3 agents with different focus areas
  - Complex research: 3-5 agents with clearly divided topics

Output Format Guidelines:
- Use clear, structured formatting
- Include executive summaries for complex topics
- Organize information logically
- Highlight key findings and insights

Remember: Quality over quantity. It's better to have fewer, more relevant sourc


SEARCH_SUBAGENT_PROMPT = """You are a specialized Search Subagent focused on fi

Your approach:
1. Start with broad searches to understand the landscape
2. Progressively narrow your focus based on findings
3. Evaluate source quality and relevance
4. Extract key information that addresses your objective

Search Strategy:
- Begin with 2-3 word queries for broader results
- Use quotes for exact phrases
- Add qualifiers progressively (year, location, type)
- Prefer primary sources over secondary when possible

Quality Criteria:
- Authoritative sources (official sites, academic papers, reputable news)
- Recent information (unless historical context needed)
- Direct relevance to the objective
- Factual, verifiable information

You have access to web search tools. Use them efficiently and stop when you hav


CITATION_AGENT_PROMPT = """You are a Citation Agent responsible for adding prop

Your tasks:
```

```
1. Review the research report and identify all claims that need citations
2. Match claims to specific sources from the provided source list
3. Insert citations in [Source N] format directly after relevant claims
4. Ensure every factual claim has at least one supporting source

Citation Guidelines:
- Place citations immediately after the claim they support
- Use multiple citations for important or controversial claims
- Prefer primary sources when available
- Include page numbers or sections when applicable

Output the same report with citations properly inserted throughout the text."""
```

Step 6: Lead Agent Implementation

The lead agent orchestrates the entire research process through distinct phases.

The planning phase uses the agent's reasoning capabilities to decompose queries into manageable subtasks, avoiding the coordination complexity mentioned in the article.

The execution phase implements parallel processing with controlled concurrency, the `MAX_PARALLEL_SUBAGENTS` limit prevents system overload.

The synthesis phase leverages the lead agent's broader context to combine findings coherently.

The iterative nature allows the system to pursue promising leads discovered during research.

The memory persistence throughout ensures the system can recover from failures without losing work.

`app/agents/lead_agent.py`:

```
from typing import List, Dict, Any, Optional
import asyncio
import json
from uuid import uuid4
import time
import re
```

```python
from app.agents.base_agent import BaseAgent
from app.agents.citation_agent import CitationAgent
from app.agents.search_agent import SearchSubAgent
from app.models.schemas import (
    ResearchQuery, ResearchPlan, SubAgentTask,
    SubAgentResult, ResearchResult, CitationInfo
)
from app.core.prompts import LEAD_AGENT_PROMPT
from app.core.config import settings
from app.tools.memory_tools import MemoryStore

class LeadResearchAgent(BaseAgent):
    """Lead agent that orchestrates the research process"""

    def __init__(self):
        super().__init__(
            model=settings.LEAD_AGENT_MODEL,
            name="Lead Research Agent"
        )
        self.memory_store = MemoryStore()
        self.active_subagents: Dict[str, SearchSubAgent] = {}

    def get_system_prompt(self) -> str:
        return LEAD_AGENT_PROMPT

    async def conduct_research(self, query: ResearchQuery) -> ResearchResult:
        """Main entry point for conducting research"""
        start_time = time.time()

        research_id = uuid4()

        # Save initial context
        await self.memory_store.save_context(
            research_id,
            {"query": query.dict(), "status": "planning"}
        )

        # Phase 1: Analyze query and create research plan
        plan = await self._create_research_plan(query)
        await self.memory_store.save_context(
            research_id,
            {"plan": plan.dict(), "status": "executing"}
        )

        # Phase 2: Execute research plan with subagents
        results = await self._execute_research_plan(plan, query.max_iterations)

        # Phase 3: Synthesize results into final report
        final_report = await self._synthesize_results(query.query, results)

        # Phase 4: Add citations (simplified for this example)
        # After adding citations, before creating ResearchResult
```

```
        cited_report = await self._add_citations(final_report, results)

        # Extract sections from report
        sections = self._extract_report_sections(cited_report)

        # Convert citation_list to CitationInfo objects
        citation_infos = [
            CitationInfo(**citation)
            for citation in self.citation_list
        ]

        # Compile final result
        all_sources = []
        for result in results:
            all_sources.extend(result.sources)

        research_result = ResearchResult(
            research_id=research_id,
            query=query.query,
            report=cited_report,
            citations=citation_infos,  # Now properly typed
            sources_used=all_sources,
            total_tokens_used=self.total_tokens + sum(r.token_count for r in re
            execution_time=time.time() - start_time,  # Add start_time tracking
            subagent_count=len(results),
            # iteration_count=iterations_used,  # Track this in execute_researc
            report_sections=sections
        )

        # Save final result
        await self.memory_store.save_result(research_id, research_result)

        return research_result

    async def _create_research_plan(self, query: ResearchQuery) -> ResearchPlan
        """Create a research plan based on the query"""

        # Use thinking to analyze the query
        thinking_result = await self.think(f"Research query: {query.query}")

        # Create planning prompt
        planning_prompt = f"""
        Create a research plan for the following query:

        Query: {query.query}

        Based on your analysis, create a detailed research plan with:
        1. Overall strategy
        2. Specific subtasks for search agents (max {query.max_subagents})
        3. Complexity assessment

        For each subtask, specify:
        - Clear objective
```

```
            - Search focus area
            - Expected output format

            Output as JSON:
            {{
                "strategy": "...",
                "complexity": "simple|moderate|complex",
                "subtasks": [
                    {{
                        "objective": "...",
                        "search_focus": "...",
                        "expected_output": "..."
                    }}
                ]
            }}
            """

            response = await self._call_llm(planning_prompt)

            # Parse response (simplified - in production use proper JSON parsing)
            try:
                plan_data = json.loads(response)

                subtasks = [
                    SubAgentTask(
                        objective=task["objective"],
                        search_focus=task["search_focus"],
                        expected_output_format=task.get("expected_output", "List of
                    )
                    for task in plan_data["subtasks"]
                ]

                return ResearchPlan(
                    strategy=plan_data["strategy"],
                    subtasks=subtasks,
                    estimated_complexity=plan_data["complexity"]
                )

            except Exception as e:
                # Fallback plan
                return ResearchPlan(
                    strategy="Direct search for information",
                    subtasks=[
                        SubAgentTask(
                            objective=f"Find information about: {query.query}",
                            search_focus=query.query,
                            expected_output_format="Relevant findings and sources"
                        )
                    ],
                    estimated_complexity="simple"
                )

        async def _execute_research_plan(
```

```
        self,
        plan: ResearchPlan,
        max_iterations: int
    ) -> List[SubAgentResult]:
        """Execute the research plan using subagents"""

        results = []
        remaining_tasks = plan.subtasks.copy()
        iteration = 0

        while remaining_tasks and iteration < max_iterations:
            iteration += 1

            # Process tasks in batches (parallel execution)
            batch_size = min(len(remaining_tasks), settings.MAX_PARALLEL_SUBAGE
            current_batch = remaining_tasks[:batch_size]
            remaining_tasks = remaining_tasks[batch_size:]

            # Create subagents for this batch
            batch_tasks = []
            for task in current_batch:
                subagent = SearchSubAgent(task_id=task.task_id)
                self.active_subagents[str(task.task_id)] = subagent
                batch_tasks.append(subagent.execute_task(task))

            # Execute batch in parallel
            batch_results = await asyncio.gather(*batch_tasks, return_exception

            # Process results
            for i, result in enumerate(batch_results):
                if isinstance(result, Exception):
                    print(f"Subagent failed: {result}")
                    # Could retry or handle error
                else:
                    results.append(result)

            # Check if we need more research based on results
            if await self._needs_more_research(results, plan.strategy):
                # Create additional tasks if needed
                new_tasks = await self._create_followup_tasks(results)
                remaining_tasks.extend(new_tasks)

        return results

    async def _needs_more_research(
        self,
        current_results: List[SubAgentResult],
        strategy: str
    ) -> bool:
        """Determine if more research is needed"""

        # Simple heuristic - in production would be more sophisticated
        if not current_results:
```

```
            return True

        total_sources = sum(len(r.sources) for r in current_results)
        avg_relevance = sum(
            sum(s.relevance_score for s in r.sources) / len(r.sources)
            for r in current_results if r.sources
        ) / len(current_results)

        # Need more research if we have few sources or low relevance
        return total_sources < 10 or avg_relevance < 0.7

    async def _create_followup_tasks(
        self,
        current_results: List[SubAgentResult]
    ) -> List[SubAgentTask]:
        """Create follow-up tasks based on current results"""

        # Analyze what we've found and what's missing
        summary = "\n".join(r.summary for r in current_results)

        prompt = f"""
        Based on the current research findings, identify gaps or areas that nee

        Current findings summary:
        {summary}

        Create up to 2 follow-up tasks that address gaps or dive deeper into im

        Output as JSON:
        {{
            "followup_tasks": [
                {{
                    "objective": "...",
                    "search_focus": "...",
                    "reason": "..."
                }}
            ]
        }}
        """

        response = await self._call_llm(prompt)

        try:
            data = json.loads(response)
            return [
                SubAgentTask(
                    objective=task["objective"],
                    search_focus=task["search_focus"],
                    expected_output_format="Detailed findings"
                )
                for task in data.get("followup_tasks", [])
            ]
        except:
```

```
            return []

    async def _synthesize_results(
        self,
        original_query: str,
        results: List[SubAgentResult]
    ) -> str:
        """Synthesize all results into a coherent report"""

        # Compile all findings
        all_findings = []
        for result in results:
            all_findings.extend(result.findings)

        synthesis_prompt = f"""
        Synthesize the following research findings into a comprehensive report.

        Original Query: {original_query}

        Research Findings:
        {json.dumps(all_findings, indent=2)}

        Create a well-structured report that:
        1. Directly answers the user's query
        2. Organizes information logically
        3. Highlights key insights
        4. Notes any limitations or gaps in the research

        Format the report with clear sections and subsections as appropriate.
        """

        report = await self._call_llm(synthesis_prompt, max_tokens=8000)
        return report

    async def _add_citations(
    self,
    report: str,
    results: List[SubAgentResult]
) -> str:
        """Add citations to the report using the Citation Agent"""

        # Initialize citation agent
        citation_agent = CitationAgent()

        # Compile all sources and findings
        all_sources = []
        all_findings = []

        for result in results:
            all_sources.extend(result.sources)
            all_findings.extend(result.findings)

        # Remove duplicate sources
```

```
        unique_sources = []
        seen_urls = set()

        for source in all_sources:
            if source.url not in seen_urls:
                seen_urls.add(source.url)
                unique_sources.append(source)

        # Add citations to the report
        cited_report, citation_list = await citation_agent.add_citations(
            report,
            unique_sources,
            all_findings
        )

        # Generate bibliography
        bibliography = await citation_agent.generate_bibliography(
            unique_sources,
            citation_list,
            style="MLA"
        )

        # Append bibliography to report
        final_report = cited_report + bibliography

        # Update the research result with citation information
        self.citation_list = citation_list

        return final_report

    def _extract_report_sections(self, report: str) -> List[str]:
        """Extract main sections from the report"""

        # Find headers (lines starting with #)
        headers = re.findall(r'^#+\s+(.+)$', report, re.MULTILINE)

        # Clean and return unique headers
        sections = []
        for header in headers:
            clean_header = header.strip()
            if clean_header and clean_header not in sections:
                sections.append(clean_header)

        return sections
```

Step 7: Search Subagent and Citation Agent Implementation

Search subagents demonstrate specialized behavior patterns.

The task execution flow mirrors human research: think about approach, search broadly, evaluate results, extract findings, and summarize.

The `_generate_search_query` method addresses the article's point about agents using overly specific queries by encouraging concise searches.

The relevance evaluation step filters out low-quality results before processing, saving tokens.

The sufficiency check prevents endless searching, agents know when to stop.

The parallel result processing within each subagent multiplies the system's efficiency.

Each method includes fallback behavior for when LLM parsing fails, ensuring robustness.

`app/agents/search_agent.py`:

```
from typing import List, Dict, Any, Optional
import asyncio
from uuid import UUID

from app.agents.base_agent import BaseAgent
from app.models.schemas import SubAgentTask, SubAgentResult, SearchResult
from app.core.prompts import SEARCH_SUBAGENT_PROMPT
from app.core.config import settings
from app.tools.search_tools import WebSearchTool

class SearchSubAgent(BaseAgent):
    """Subagent specialized in searching for specific information"""

    def __init__(self, task_id: UUID):
        super().__init__(
            model=settings.SUBAGENT_MODEL,
            name=f"Search Subagent {task_id}"
        )
        self.task_id = task_id
        self.search_tool = WebSearchTool()

    def get_system_prompt(self) -> str:
        return SEARCH_SUBAGENT_PROMPT

    async def execute_task(self, task: SubAgentTask) -> SubAgentResult:
        """Execute the assigned research task"""
```

```
        # Think about approach
        thinking = await self.think(
            f"Task: {task.objective}\nFocus: {task.search_focus}"
        )

        # Execute searches based on plan
        all_results = []
        findings = []

        for step in thinking.get("steps", [])[:task.max_searches]:
            # Generate search query
            query = await self._generate_search_query(
                task.objective,
                task.search_focus,
                step,
                all_results
            )

            # Perform search
            search_results = await self.search_tool.search(query)

            # Evaluate results
            relevant_results = await self._evaluate_results(
                search_results,
                task.objective
            )

            all_results.extend(relevant_results)

            # Extract findings from relevant results
            extracted = await self._extract_findings(
                relevant_results,
                task.objective,
                task.expected_output_format
            )

            findings.extend(extracted)

            # Check if we have enough information
            if await self._has_sufficient_information(findings, task.objective)
                break

        # Summarize findings
        summary = await self._summarize_findings(findings, task.objective)

        return SubAgentResult(
            task_id=self.task_id,
            findings=findings,
            sources=all_results,
            summary=summary,
            token_count=self.total_tokens
        )
```

```python
    async def _generate_search_query(
        self,
        objective: str,
        focus: str,
        step: str,
        previous_results: List[SearchResult]
    ) -> str:
        """Generate an effective search query"""

        prompt = f"""
        Generate a search query for the following:

        Objective: {objective}
        Focus Area: {focus}
        Current Step: {step}

        Previous searches found {len(previous_results)} results.

        Create a search query that:
        - Is concise (2-5 words preferred)
        - Targets the specific information needed
        - Avoids redundancy with previous searches

        Output only the search query, nothing else.
        """

        query = await self._call_llm(prompt, max_tokens=100)
        return query.strip()

    async def _evaluate_results(
        self,
        results: List[SearchResult],
        objective: str
    ) -> List[SearchResult]:
        """Evaluate search results for relevance"""

        if not results:
            return []

        # Create evaluation prompt
        results_summary = "\n".join([
            f"{i+1}. {r.title} - {r.snippet[:200]}..."
            for i, r in enumerate(results[:10])
        ])

        prompt = f"""
        Evaluate these search results for relevance to the objective.

        Objective: {objective}

        Search Results:
        {results_summary}
```

```
        For each result, rate its relevance from 0.0 to 1.0.
        Consider:
        - Direct relevance to the objective
        - Quality of the source
        - Uniqueness of information

        Output as JSON:
        {{
            "evaluations": [
                {{"index": 1, "relevance": 0.9, "reason": "..."}}
            ]
        }}
        """

        response = await self._call_llm(prompt, max_tokens=2000)

        # Parse evaluations and update relevance scores
        try:
            import json
            data = json.loads(response)

            for eval in data.get("evaluations", []):
                idx = eval["index"] - 1
                if 0 <= idx < len(results):
                    results[idx].relevance_score = eval["relevance"]

            # Return only relevant results
            return [r for r in results if r.relevance_score >= 0.6]

        except:
            # If parsing fails, return top results
            return results[:5]

    async def _extract_findings(
        self,
        results: List[SearchResult],
        objective: str,
        output_format: str
    ) -> List[Dict[str, Any]]:
        """Extract key findings from search results"""

        if not results:
            return []

        # Compile content from results
        content_summary = "\n\n".join([
            f"Source: {r.title}\nURL: {r.url}\nContent: {r.snippet}"
            for r in results
        ])

        prompt = f"""
        Extract key findings from these sources related to the objective.
```

```
        Objective: {objective}
        Expected Format: {output_format}

        Sources:
        {content_summary}

        Extract specific, factual findings that address the objective.

        Output as JSON:
        {{
            "findings": [
                {{
                    "fact": "...",
                    "source_url": "...",
                    "source_title": "...",
                    "relevance": "..."
                }}
            ]
        }}
        """

        response = await self._call_llm(prompt, max_tokens=3000)

        try:
            import json
            data = json.loads(response)
            return data.get("findings", [])
        except:
            return []

    async def _has_sufficient_information(
        self,
        findings: List[Dict[str, Any]],
        objective: str
    ) -> bool:
        """Determine if we have enough information"""

        if len(findings) < 3:
            return False

        prompt = f"""
        Assess if we have sufficient information to address the objective.

        Objective: {objective}

        We have found {len(findings)} pieces of information.

        Do we have enough high-quality, relevant information to comprehensively
        Answer with YES or NO and a brief reason.
        """

        response = await self._call_llm(prompt, max_tokens=200)
```

```
            return "YES" in response.upper()

        async def _summarize_findings(
            self,
            findings: List[Dict[str, Any]],
            objective: str
        ) -> str:
            """Create a summary of all findings"""

            findings_text = "\n".join([
                f"- {f.get('fact', 'Unknown')} (Source: {f.get('source_title', 'Unk
                for f in findings
            ])

            prompt = f"""
            Summarize these research findings in relation to the objective.

            Objective: {objective}

            Findings:
            {findings_text}

            Create a concise summary (2-3 paragraphs) that:
            1. Addresses the main objective
            2. Highlights the most important findings
            3. Notes any gaps or limitations
            """

            summary = await self._call_llm(prompt, max_tokens=1000)
            return summary
```

`app/agents/citation_agent.py`:

```
from typing import List, Dict, Any, Tuple, Optional
import re
import json
from dataclasses import dataclass

from app.agents.base_agent import BaseAgent
from app.models.schemas import SearchResult
from app.core.prompts import CITATION_AGENT_PROMPT
from app.core.config import settings


@dataclass
class Citation:
    """Represents a citation to be inserted"""
```

```
    claim_text: str
    source_index: int
    source_url: str
    source_title: str
    confidence: float
    position: int  # Character position in text


class CitationAgent(BaseAgent):
    """Agent responsible for adding citations to research reports"""

    def __init__(self):
        super().__init__(
            model=settings.CITATION_MODEL,
            name="Citation Agent"
        )

    def get_system_prompt(self) -> str:
        return CITATION_AGENT_PROMPT

    async def add_citations(
        self,
        report: str,
        sources: List[SearchResult],
        findings: List[Dict[str, Any]] = None
    ) -> Tuple[str, List[Dict[str, Any]]]:
        """
        Add citations to a research report

        Args:
            report: The research report text
            sources: List of sources used in the research
            findings: Optional list of specific findings with their sources

        Returns:
            Tuple of (cited_report, citation_list)
        """

        # First, create a source index
        source_index = self._create_source_index(sources)

        # Identify claims that need citations
        claims = await self._identify_claims(report)

        # Match claims to sources
        citations = await self._match_claims_to_sources(
            claims,
            sources,
            findings
        )

        # Insert citations into the report
        cited_report = await self._insert_citations(report, citations)
```

```
        # Generate citation list
        citation_list = self._generate_citation_list(citations, source_index)

        return cited_report, citation_list

    def _create_source_index(self, sources: List[SearchResult]) -> Dict[int, Se
        """Create an index of sources for easy reference"""
        return {i + 1: source for i, source in enumerate(sources)}

    async def _identify_claims(self, report: str) -> List[Dict[str, Any]]:
        """Identify factual claims in the report that need citations"""

        # Split report into sentences for analysis
        sentences = self._split_into_sentences(report)

        # Batch sentences for efficient processing
        batch_size = 20
        all_claims = []

        for i in range(0, len(sentences), batch_size):
            batch = sentences[i:i + batch_size]
            batch_text = "\n".join([f"{j+1}. {sent}" for j, sent in enumerate(b

            prompt = f"""
            Identify factual claims in these sentences that require citations.

            Sentences:
            {batch_text}

            For each sentence containing a factual claim, identify:
            1. The sentence number
            2. The specific claim that needs citation
            3. The type of claim (statistic, fact, quote, finding, comparison)
            4. How important citation is (high/medium/low)

            Skip:
            - General knowledge or common facts
            - Transitional sentences
            - Questions or hypotheticals
            - Section headers

            Output as JSON:
            {{
                "claims": [
                    {{
                        "sentence_num": 1,
                        "text": "...",
                        "claim": "specific claim text",
                        "type": "statistic|fact|quote|finding|comparison",
                        "importance": "high|medium|low"
                    }}
                ]
```

```
            }}
            """

            response = await self._call_llm(prompt, max_tokens=2000)

            try:
                data = json.loads(response)

                # Adjust sentence numbers to global position
                for claim in data.get("claims", []):
                    claim["sentence_num"] = i + claim["sentence_num"] - 1
                    claim["text"] = sentences[claim["sentence_num"]]

                all_claims.extend(data.get("claims", []))

            except Exception as e:
                print(f"Error parsing claims: {e}")
                continue

        return all_claims

    async def _match_claims_to_sources(
        self,
        claims: List[Dict[str, Any]],
        sources: List[SearchResult],
        findings: List[Dict[str, Any]] = None
    ) -> List[Citation]:
        """Match identified claims to appropriate sources"""

        citations = []

        # If we have findings with explicit source mappings, use those first
        finding_map = {}
        if findings:
            for finding in findings:
                fact = finding.get("fact", "")
                source_url = finding.get("source_url", "")
                if fact and source_url:
                    finding_map[fact.lower()] = source_url

        # Process claims in batches
        for claim in claims:
            # First check if this claim matches a known finding
            claim_text = claim["claim"].lower()
            matched_source = None

            # Check finding map
            for fact_text, source_url in finding_map.items():
                if self._text_similarity(claim_text, fact_text) > 0.8:
                    # Find the source index
                    for i, source in enumerate(sources):
                        if source.url == source_url:
                            matched_source = (i + 1, source, 0.9)
```

```
                        break
                break

        # If no direct match, search all sources
        if not matched_source:
            source_matches = await self._find_best_source_match(
                claim,
                sources
            )
            if source_matches:
                matched_source = source_matches[0]

        # Create citation if match found
        if matched_source:
            source_idx, source, confidence = matched_source

            # Find position in original text
            position = self._find_text_position(
                claim["text"],
                claims[0]["text"] if claims else ""
            )

            citations.append(Citation(
                claim_text=claim["text"],
                source_index=source_idx,
                source_url=source.url,
                source_title=source.title,
                confidence=confidence,
                position=position
            ))

    return citations

async def _find_best_source_match(
    self,
    claim: Dict[str, Any],
    sources: List[SearchResult]
) -> List[Tuple[int, SearchResult, float]]:
    """Find the best source match for a claim"""

    # Create a batch prompt to evaluate all sources at once
    sources_summary = "\n\n".join([
        f"Source {i+1}:\nTitle: {s.title}\nURL: {s.url}\nContent: {s.snippe
        for i, s in enumerate(sources)
    ])

    prompt = f"""
    Match this claim to the most appropriate source.

    Claim: "{claim['claim']}"
    Type: {claim['type']}
    Full sentence: "{claim['text']}"
```

```
        Available sources:
        {sources_summary}

        Evaluate which sources best support this claim. Consider:
        1. Direct mention of the fact/statistic
        2. Relevance to the claim
        3. Authority of the source
        4. Specificity of the match

        Output as JSON:
        {{
            "matches": [
                {{
                    "source_num": 1,
                    "confidence": 0.9,
                    "reason": "..."
                }}
            ]
        }}

        Only include sources with confidence > 0.6.
        """

        response = await self._call_llm(prompt, max_tokens=1000)

        try:
            data = json.loads(response)
            matches = []

            for match in data.get("matches", []):
                source_idx = match["source_num"]
                if 1 <= source_idx <= len(sources):
                    matches.append((
                        source_idx,
                        sources[source_idx - 1],
                        match["confidence"]
                    ))

            # Sort by confidence
            matches.sort(key=lambda x: x[2], reverse=True)
            return matches

        except Exception as e:
            print(f"Error parsing source matches: {e}")
            return []

    async def _insert_citations(
        self,
        report: str,
        citations: List[Citation]
    ) -> str:
        """Insert citations into the report text"""
```

```
        # Sort citations by position (reverse order to maintain positions)
        citations.sort(key=lambda x: x.position, reverse=True)

        # Group citations by claim to handle multiple sources
        claim_citations = {}
        for citation in citations:
            key = citation.claim_text
            if key not in claim_citations:
                claim_citations[key] = []
            claim_citations[key].append(citation)

        # Process the report
        cited_report = report
        processed_claims = set()

        for claim_text, cite_list in claim_citations.items():
            if claim_text in processed_claims:
                continue

            # Find all occurrences of this claim in the report
            pattern = re.escape(claim_text)
            matches = list(re.finditer(pattern, cited_report))

            if not matches:
                # Try finding partial match
                sentences = self._split_into_sentences(cited_report)
                for i, sentence in enumerate(sentences):
                    if self._text_similarity(sentence, claim_text) > 0.8:
                        # Found similar sentence
                        matches = list(re.finditer(re.escape(sentence), cited_r
                        if matches:
                            claim_text = sentence
                            break

            # Insert citations after each occurrence
            for match in reversed(matches):  # Reverse to maintain positions
                end_pos = match.end()

                # Build citation string
                if len(cite_list) == 1:
                    citation_str = f"[{cite_list[0].source_index}]"
                else:
                    # Multiple sources
                    indices = sorted(set(c.source_index for c in cite_list))
                    citation_str = "[" + ",".join(str(i) for i in indices) + "]

                # Insert citation
                cited_report = (
                    cited_report[:end_pos] +
                    " " + citation_str +
                    cited_report[end_pos:]
                )
```

```
                processed_claims.add(claim_text)

        return cited_report

    def _generate_citation_list(
        self,
        citations: List[Citation],
        source_index: Dict[int, SearchResult]
    ) -> List[Dict[str, Any]]:
        """Generate a formatted citation list"""

        # Get unique sources that were cited
        cited_indices = sorted(set(c.source_index for c in citations))

        citation_list = []
        for idx in cited_indices:
            source = source_index.get(idx)
            if source:
                citation_list.append({
                    "index": idx,
                    "title": source.title,
                    "url": source.url,
                    "times_cited": sum(1 for c in citations if c.source_index =
                })

        return citation_list

    def _split_into_sentences(self, text: str) -> List[str]:
        """Split text into sentences"""
        # Simple sentence splitter - in production use nltk or spacy
        sentences = re.split(r'(?<=[.!?])\s+', text)

        # Clean up sentences
        cleaned = []
        for sent in sentences:
            sent = sent.strip()
            if sent and len(sent) > 10:  # Skip very short fragments
                cleaned.append(sent)

        return cleaned

    def _text_similarity(self, text1: str, text2: str) -> float:
        """Calculate simple text similarity (0-1)"""
        # Simple implementation - in production use better similarity metrics
        text1_lower = text1.lower()
        text2_lower = text2.lower()

        if text1_lower == text2_lower:
            return 1.0

        # Check if one contains the other
        if text1_lower in text2_lower or text2_lower in text1_lower:
            return 0.8
```

```
        # Count common words
        words1 = set(text1_lower.split())
        words2 = set(text2_lower.split())

        if not words1 or not words2:
            return 0.0

        common = len(words1.intersection(words2))
        total = len(words1.union(words2))

        return common / total if total > 0 else 0.0

    def _find_text_position(self, text: str, reference: str) -> int:
        """Find the position of text in the report"""
        # Simple position finder
        try:
            return reference.index(text)
        except ValueError:
            return 0

    async def generate_bibliography(
        self,
        sources: List[SearchResult],
        citation_list: List[Dict[str, Any]],
        style: str = "MLA"
    ) -> str:
        """Generate a formatted bibliography"""

        bibliography = "\n\n## References\n\n"

        for citation in citation_list:
            idx = citation["index"]
            source = next((s for s in sources if s.url == citation["url"]), Non

            if source:
                if style == "MLA":
                    # Simple MLA-style citation
                    entry = f"[{idx}] \"{source.title}.\" Web. {source.url}"
                elif style == "APA":
                    # Simple APA-style citation
                    entry = f"[{idx}] {source.title}. Retrieved from {source.ur
                else:
                    # Default simple format
                    entry = f"[{idx}] {source.title}. {source.url}"

                bibliography += entry + "\n\n"

        return bibliography
```

Step 8: Search Tools Implementation

The search tool abstraction allows easy swapping of search providers. The mock implementation facilitates testing without API costs.

In production, this would integrate with real search APIs, handling rate limits and failures gracefully.

The parallel content fetching demonstrates how tools can internally optimize performance.

The enrichment pattern, fetching full content for promising results, balances thoroughness with efficiency.

The error handling ensures that one failed fetch doesn't break the entire search batch.

The resource cleanup in the `close` method prevents connection leaks in long-running systems.

`app/tools/search_tools.py`:

```
import asyncio
import httpx
from typing import List, Dict, Any, Optional
from bs4 import BeautifulSoup
import json

from app.models.schemas import SearchResult
from app.core.config import settings

class WebSearchTool:
    """Tool for performing web searches"""

    def __init__(self):
        self.client = httpx.AsyncClient(timeout=settings.SEARCH_TIMEOUT)

    async def search(self, query: str) -> List[SearchResult]:
        """
        Perform a web search and return results

        Note: This is a mock implementation. In production, you would:
        1. Use a real search API (Google, Bing, etc.)
        2. Implement proper rate limiting
        3. Handle API errors gracefully
```

```
        """

        # Mock search results for demonstration
        # In production, replace with actual API calls
        mock_results = await self._mock_search(query)

        # Process results in parallel
        tasks = [self._fetch_content(result) for result in mock_results]
        enriched_results = await asyncio.gather(*tasks, return_exceptions=True)

        # Filter out failed fetches
        valid_results = [
            r for r in enriched_results
            if isinstance(r, SearchResult)
        ]

        return valid_results

    async def _mock_search(self, query: str) -> List[Dict[str, Any]]:
        """Mock search function for demonstration"""

        # In production, this would call a real search API
        # For now, return mock data based on query keywords

        base_results = [
            {
                "url": f"https://example.com/ai-agents-{i}",
                "title": f"AI Agents in 2025: {query} - Result {i}",
                "snippet": f"This article discusses {query} and how AI agents a
            }
            for i in range(1, 6)
        ]

        return base_results

    async def _fetch_content(self, result: Dict[str, Any]) -> SearchResult:
        """Fetch and parse content from a URL"""

        try:
            # In production, actually fetch the URL
            # For mock, just create a SearchResult

            return SearchResult(
                url=result["url"],
                title=result["title"],
                snippet=result["snippet"],
                content=f"Full content about {result['title']}...",  # Mock con
                relevance_score=0.8  # Will be updated by agent
            )

        except Exception as e:
            print(f"Error fetching {result.get('url', 'unknown')}: {e}")
            raise
```

```
    async def close(self):
        """Clean up resources"""
        await self.client.aclose()
```

Step 9: Memory Store Implementation

The memory store addresses several distributed system challenges. The TTL-based expiration prevents unbounded memory growth while keeping recent context available.

The key namespacing (`context:` and `result:` prefixes) allows different data types to coexist.

In production, this would use Redis for persistence across restarts and scaling across multiple servers.

The async interface ensures memory operations don't block agent execution.

The separation between context (intermediate state) and results (final output) allows different retention policies.

The cleanup method would typically run as a background task to maintain system health.

`app/tools/memory_tools.py`:

```
import json
import asyncio
from typing import Dict, Any, Optional
from datetime import datetime, timedelta
from uuid import UUID

from app.models.schemas import ResearchResult
from app.core.config import settings

class MemoryStore:
    """
    In-memory store for agent context and results

    In production, this would use Redis or another persistent store
```

```
    """

    def __init__(self):
        self._store: Dict[str, Dict[str, Any]] = {}
        self._ttl: Dict[str, datetime] = {}

    async def save_context(self, research_id: UUID, context: Dict[str, Any]):
        """Save research context"""
        key = f"context:{research_id}"
        self._store[key] = context
        self._ttl[key] = datetime.utcnow() + timedelta(seconds=settings.MEMORY_

    async def get_context(self, research_id: UUID) -> Optional[Dict[str, Any]]:
        """Retrieve research context"""
        key = f"context:{research_id}"

        # Check if expired
        if key in self._ttl and datetime.utcnow() > self._ttl[key]:
            del self._store[key]
            del self._ttl[key]
            return None

        return self._store.get(key)

    async def save_result(self, research_id: UUID, result: ResearchResult):
        """Save final research result"""
        key = f"result:{research_id}"
        self._store[key] = result.dict()
        self._ttl[key] = datetime.utcnow() + timedelta(seconds=settings.MEMORY_

    async def get_result(self, research_id: UUID) -> Optional[ResearchResult]:
        """Retrieve research result"""
        key = f"result:{research_id}"

        if key in self._ttl and datetime.utcnow() > self._ttl[key]:
            del self._store[key]
            del self._ttl[key]
            return None

        data = self._store.get(key)
        if data:
            return ResearchResult(**data)
        return None

    async def cleanup_expired(self):
        """Remove expired entries"""
        now = datetime.utcnow()
        expired_keys = [
            key for key, expiry in self._ttl.items()
            if now > expiry
        ]

        for key in expired_keys:
```

```
            del self._store[key]
            del self._ttl[key]
```

Step 10: Research Service

The service layer provides a clean API for the FastAPI endpoints while managing complex agent lifecycles.

Task tracking enables status monitoring without blocking the API. The service pattern also provides a place for cross-cutting concerns like authentication, rate limiting, and usage tracking.

The separation between starting research and retrieving results reflects the async nature of agent work, clients poll for results rather than blocking.

This pattern scales better than synchronous processing and provides better user experience for long-running research tasks.

`app/services/research_service.py`:

```
from typing import Optional, Dict, Any
from uuid import UUID
import asyncio
import time

from app.models.schemas import ResearchQuery, ResearchResult
from app.agents.lead_agent import LeadResearchAgent
from app.tools.memory_tools import MemoryStore

class ResearchService:
    """Service layer for research operations"""

    def __init__(self):
        self.memory_store = MemoryStore()
        self._active_research: Dict[UUID, asyncio.Task] = {}

    async def start_research(self, query: ResearchQuery) -> UUID:
        """Start a new research task"""

        # Create lead agent
        lead_agent = LeadResearchAgent()

        # Start research task
```

```
            task = asyncio.create_task(
                lead_agent.conduct_research(query)
            )

            # Track active research
            research_id = UUID()  # This will be set by the agent
            self._active_research[research_id] = task

            return research_id

        async def get_research_status(self, research_id: UUID) -> Dict[str, Any]:
            """Get the status of a research task"""

            # Check if research is still active
            if research_id in self._active_research:
                task = self._active_research[research_id]
                if not task.done():
                    # Still running
                    context = await self.memory_store.get_context(research_id)
                    return {
                        "status": context.get("status", "unknown") if context else
                        "message": "Research in progress"
                    }

            # Check for completed result
            result = await self.memory_store.get_result(research_id)
            if result:
                return {
                    "status": "completed",
                    "result": result
                }

            return {
                "status": "not_found",
                "message": "Research ID not found"
            }

        async def get_research_result(self, research_id: UUID) -> Optional[Research
            """Get the result of a completed research task"""
            return await self.memory_store.get_result(research_id)
```

Step 11: FastAPI Application

The API design follows RESTful principles while accommodating the async nature of agent systems.

The /research/start endpoint immediately returns a tracking ID, allowing clients to monitor progress without holding connections open.

The status endpoint provides visibility into the research process, crucial for user experience during long operations.

The demo endpoint serves both testing and user onboarding purposes.

The available tools endpoint enables dynamic tool discovery, important as the system grows.

Global exception handling ensures clean error responses even when unexpected failures occur.

The startup/shutdown hooks provide proper resource initialization and cleanup.

`app/main.py`:

```
from fastapi import FastAPI, HTTPException, BackgroundTasks
from fastapi.responses import JSONResponse
from typing import Dict, Any
from uuid import UUID
import asyncio

from app.agents.lead_agent import LeadResearchAgent
from app.agents.citation_agent import CitationAgent
from app.models.schemas import ResearchQuery, ResearchResult, SearchResult
from app.services.research_service import ResearchService
from app.core.config import settings

# Initialize FastAPI app
app = FastAPI(
    title="Multi-Agent Research System",
    description="A multi-agent system for conducting comprehensive research",
    version="1.0.0"
)

# Initialize research service
research_service = ResearchService()

@app.get("/")
async def root():
    """Health check endpoint"""
    return {
        "status": "healthy",
        "service": "Multi-Agent Research System",
```

```
        "version": "1.0.0"
    }

@app.post("/research/start")
async def start_research(
    query: ResearchQuery,
    background_tasks: BackgroundTasks
) -> Dict[str, Any]:
    """
    Start a new research task

    This endpoint initiates a research process that runs asynchronously.
    Returns a research_id that can be used to check status and retrieve results
    """

    try:
        # Validate query
        if not query.query.strip():
            raise HTTPException(status_code=400, detail="Query cannot be empty"

        # Start research asynchronously
        async def run_research():
            lead_agent = LeadResearchAgent()
            result = await lead_agent.conduct_research(query)
            return result

        # Create task
        task = asyncio.create_task(run_research())

        # Generate research ID (in production, this would be handled differentl
        research_id = UUID('12345678-1234-5678-1234-567812345678')  # Mock ID

        return {
            "research_id": str(research_id),
            "status": "started",
            "message": "Research task initiated successfully"
        }

    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

@app.get("/research/{research_id}/status")
async def get_research_status(research_id: UUID) -> Dict[str, Any]:
    """
    Get the status of a research task

    Returns the current status and any available intermediate results
    """

    status = await research_service.get_research_status(research_id)

    if status["status"] == "not_found":
        raise HTTPException(status_code=404, detail="Research ID not found")
```

```
    return status

@app.get("/research/{research_id}/result")
async def get_research_result(research_id: UUID) -> ResearchResult:
    """
    Get the final result of a completed research task

    Returns the full research report with citations and sources
    """

    result = await research_service.get_research_result(research_id)

    if not result:
        raise HTTPException(
            status_code=404,
            detail="Research result not found or not yet completed"
        )

    return result

@app.post("/research/demo")
async def demo_research() -> Dict[str, Any]:
    """
    Run a demo research task synchronously for testing

    This endpoint is for demonstration purposes and runs a simplified research
    """

    # Create a demo query
    demo_query = ResearchQuery(
        query="What are the top 5 AI agent companies in 2025?",
        max_subagents=2,
        max_iterations=2
    )

    # Run research synchronously for demo
    lead_agent = LeadResearchAgent()

    # For demo, we'll return a mock result
    return {
        "status": "completed",
        "demo_result": {
            "query": demo_query.query,
            "report": """
# Top 5 AI Agent Companies in 2025

Based on our research, here are the leading companies in the AI agent space:

## 1. Anthropic
- **Product**: Claude AI Assistant
- **Focus**: Multi-agent systems for research and analysis
- **Industry**: General AI, Research
```

```
## 2. OpenAI
- **Product**: GPT Agents
- **Focus**: Autonomous agents for various tasks
- **Industry**: General AI, Enterprise

## 3. Google DeepMind
- **Product**: Gemini Agents
- **Focus**: Specialized agents for specific domains
- **Industry**: General AI, Research

## 4. Microsoft
- **Product**: Copilot Agents
- **Focus**: Productivity and enterprise agents
- **Industry**: Enterprise Software

## 5. Cohere
- **Product**: Command Agents
- **Focus**: Business process automation
- **Industry**: Enterprise AI

These companies are leading the development of sophisticated AI agent systems t
            """,
            "sources_count": 15,
            "tokens_used": 5000
        }
    }

@app.post("/research/test-citations")
async def test_citations() -> Dict[str, Any]:
    """
    Test the citation agent functionality

    This endpoint demonstrates how the citation agent adds citations to a repor
    """

    # Sample report without citations
    sample_report = """
    # AI Agents in 2025: A Comprehensive Overview

    The AI agent landscape has evolved dramatically in 2025. Major companies li

    ## Market Growth

    The global AI agents market reached $15.2 billion in 2025, representing a 1

    ## Technical Advances

    Claude 3.5 introduced revolutionary multi-agent coordination capabilities i

    ## Key Players

    Anthropic leads in research and analysis applications with a 32% market sha
```

```
## Future Outlook

Industry analysts predict the AI agent market will reach $50 billion by 202
"""

# Sample sources
sample_sources = [
    SearchResult(
        url="https://techcrunch.com/2025/ai-agents-market-report",
        title="AI Agents Market Reaches $15.2 Billion in 2025",
        snippet="The global AI agents market has experienced explosive grow
        relevance_score=0.95
    ),
    SearchResult(
        url="https://anthropic.com/blog/claude-3-5-launch",
        title="Introducing Claude 3.5: Revolutionary Multi-Agent Coordinati
        snippet="Claude 3.5 introduces groundbreaking multi-agent coordinat
        relevance_score=0.98
    ),
    SearchResult(
        url="https://fortune.com/2025/enterprise-ai-adoption",
        title="73% of Fortune 500 Companies Now Use AI Agents",
        snippet="A new study reveals that 73% of Fortune 500 companies have
        relevance_score=0.92
    ),
    SearchResult(
        url="https://gartner.com/ai-market-analysis-2025",
        title="Gartner: AI Agent Market Analysis and Predictions",
        snippet="Gartner analysts predict the AI agent market will reach $5
        relevance_score=0.89
    ),
    SearchResult(
        url="https://openai.com/blog/gpt-5-agents",
        title="GPT-5 Agents: Focused on Developer Productivity",
        snippet="OpenAI's GPT-5 agents have captured 41% of the developer t
        relevance_score=0.94
    )
]

# Sample findings that map facts to sources
sample_findings = [
    {
        "fact": "The global AI agents market reached $15.2 billion in 2025"
        "source_url": "https://techcrunch.com/2025/ai-agents-market-report"
        "source_title": "AI Agents Market Reaches $15.2 Billion in 2025"
    },
    {
        "fact": "73% of Fortune 500 companies now using some form of AI age
        "source_url": "https://fortune.com/2025/enterprise-ai-adoption",
        "source_title": "73% of Fortune 500 Companies Now Use AI Agents"
    },
    {
```

```
            "fact": "Claude 3.5 can orchestrate up to 50 specialized agents",
            "source_url": "https://anthropic.com/blog/claude-3-5-launch",
            "source_title": "Introducing Claude 3.5: Revolutionary Multi-Agent
        },
        {
            "fact": "Anthropic leads with a 32% market share",
            "source_url": "https://gartner.com/ai-market-analysis-2025",
            "source_title": "Gartner: AI Agent Market Analysis and Predictions"
        },
        {
            "fact": "OpenAI dominates with 41% market share in developer tools"
            "source_url": "https://openai.com/blog/gpt-5-agents",
            "source_title": "GPT-5 Agents: Focused on Developer Productivity"
        }
    ]

    # Initialize citation agent
    citation_agent = CitationAgent()

    # Add citations
    cited_report, citation_list = await citation_agent.add_citations(
        sample_report,
        sample_sources,
        sample_findings
    )

    # Generate bibliography
    bibliography = await citation_agent.generate_bibliography(
        sample_sources,
        citation_list
    )

    return {
        "original_report_length": len(sample_report),
        "cited_report_length": len(cited_report),
        "citations_added": len(citation_list),
        "citation_list": citation_list,
        "sample_of_cited_report": cited_report[:1000] + "...",
        "full_cited_report": cited_report + bibliography
    }

@app.get("/tools/available")
async def get_available_tools() -> Dict[str, Any]:
    """Get list of available tools for agents"""

    return {
        "tools": [
            {
                "name": "web_search",
                "description": "Search the web for information",
                "parameters": ["query", "max_results"]
            },
            {
```

```
                "name": "memory_store",
                "description": "Store and retrieve context",
                "parameters": ["key", "value"]
            }
        ]
    }

# Error handlers
@app.exception_handler(Exception)
async def general_exception_handler(request, exc):
    return JSONResponse(
        status_code=500,
        content={
            "detail": "An internal error occurred",
            "type": type(exc).__name__
        }
    )

# Startup and shutdown events
@app.on_event("startup")
async def startup_event():
    """Initialize services on startup"""
    print("Multi-Agent Research System starting up...")

@app.on_event("shutdown")
async def shutdown_event():
    """Cleanup on shutdown"""
    print("Multi-Agent Research System shutting down...")
```

Step 12: Running the Application

Create a `.env` file:

```
ANTHROPIC_API_KEY=your_api_key_here
REDIS_URL=redis://localhost:6379
```

Create a startup script `run.py`:

```
import uvicorn

if __name__ == "__main__":
    uvicorn.run(
```

```
        "app.main:app",
        host="0.0.0.0",
        port=8000,
        reload=True,
        log_level="info"
    )
```

Step 13: Testing the System

The test client demonstrates proper interaction patterns with an async multi-agent system.

It shows how to handle the asynchronous nature of research tasks through polling.

The progressive test approach, from health checks to demo to real research, helps isolate issues during development.

The status polling with backoff prevents overwhelming the server while providing timely updates.

This testing pattern can be extended into integration tests and load tests for production validation.

Create a test client `test_client.py`:

```
import asyncio
from app.models.schemas import ResearchQuery, SubAgentTask, SearchResult
from app.agents.lead_agent import LeadResearchAgent
from app.agents.search_agent import SearchSubAgent
from app.agents.citation_agent import CitationAgent


async def test_full_research_pipeline():
    """Test the complete multi-agent research pipeline"""

    print("=== Testing Multi-Agent Research System ===\n")

    # 1. Test Citation Agent independently
    print("1. Testing Citation Agent...")

    citation_agent = CitationAgent()

    test_report = """
    Recent studies show that AI adoption in healthcare has increased by 45% in
```

```
        Major hospitals are using AI for diagnosis, with accuracy rates reaching 94
        The FDA has approved 23 new AI-powered medical devices this year.
        """

        test_sources = [
            SearchResult(
                url="https://healthtech.com/ai-adoption-2025",
                title="Healthcare AI Adoption Surges 45% in 2025",
                snippet="A comprehensive study reveals that AI adoption in healthca
                relevance_score=0.95
            ),
            SearchResult(
                url="https://fda.gov/ai-medical-devices-2025",
                title="FDA Approves 23 AI-Powered Medical Devices in 2025",
                snippet="The FDA has approved 23 new AI-powered medical devices thi
                relevance_score=0.93
            )
        ]

        cited_report, citations = await citation_agent.add_citations(test_report, t
        print(f"✓ Citation Agent added {len(citations)} citations\n")

        # 2. Test Search Subagent
        print("2. Testing Search Subagent...")

        test_task = SubAgentTask(
            objective="Find information about AI agents in healthcare",
            search_focus="AI medical diagnosis accuracy statistics 2025",
            expected_output_format="List of statistics with sources"
        )

        search_agent = SearchSubAgent(test_task.task_id)
        result = await search_agent.execute_task(test_task)

        print(f"✓ Search Agent found {len(result.sources)} sources")
        print(f"✓ Extracted {len(result.findings)} findings\n")

        # 3. Test Full Pipeline
        print("3. Testing Complete Research Pipeline...")

        research_query = ResearchQuery(
            query="What are the latest breakthroughs in AI-powered medical diagnosi
            max_subagents=2,
            max_iterations=2
        )

        lead_agent = LeadResearchAgent()
        research_result = await lead_agent.conduct_research(research_query)

        print(f"✓ Research completed!")
        print(f"✓ Total tokens used: {research_result.total_tokens_used}")
        print(f"✓ Sources found: {len(research_result.sources_used)}")
        print(f"✓ Report length: {len(research_result.report)} characters")
```

```
        # Show a sample of the report with citations
        print("\n=== Sample of Final Report ===")
        print(research_result.report[:500] + "...")

        # Check that citations were added
        citation_pattern = r'\[\d+\]'
        import re
        citations_found = len(re.findall(citation_pattern, research_result.report))
        print(f"\n✓ Citations in report: {citations_found}")

        print("\n=== Test Complete ===")


    if __name__ == "__main__":
        asyncio.run(test_full_research_pipeline())
```

Running the System

1. Start the FastAPI server:

```
python3 run.py
```

2. In another terminal, run the test client:

```
python3 test_client.py
```

3. Access the interactive API documentation:

- Open http://localhost:8000/docs in your browser

You can go ahead and test the /research/demo endpoint,

which will output the following response body


```
{
  "status": "completed",
  "demo_result": {
    "query": "What are the top 5 AI agent companies in 2025?",
    "report": "\n# Top 5 AI Agent Companies in 2025\n\nBased on our research, h
    "sources_count": 15,
    "tokens_used": 5000
  }
}
```


Closing Thoughts on Implementation

When building intelligent agent systems, success often lies in the choices that shape how the whole system performs, scales, and recovers from failure.

Here are some of the key lessons and patterns we have come to rely on.

- **Prompt engineering:** turned out to be one of the foundational pillars. Each agent starts its life with a sharply focused prompt, a clear, concise instruction set that guides how it interacts with tools and solves problems.
- When it comes to **execution**, speed matters, but control matters more. That's we leaned into **parallelism**, spinning up subagents concurrently using `asyncio`. This allows them to handle tasks in batches, without overwhelming the system. Of course, not every subagent behaves, so we built in robust error handling to make sure a single failure doesn't bring the whole process down.
- **State management** is another area where things often get messy unless handled with care. To ensure continuity in long-running tasks, weset up a memory store that persists the context between steps. Research plans get stored for recovery, and results are cached, so agents don't have to repeat work they've already done. It's like giving your system a reliable short-term memory.
- Now, **errors** are inevitable. Tools fail, APIs flake out, and weird edge cases pop up. That's just reality. The goal isn't to avoid failure entirely, it's to fail gracefully. We added retry logic for transient issues and crafted clear error messages that actually help when debugging. It's the kind of work that's invisible when things go right, but priceless when they don't.
- Observability also became a non-negotiable. We built in **token counters** to track costs, **execution timers** to monitor performance, and clean, human-readable **status reports** to keep tabs on what's happening under the hood. Without that visibility, you're flying blind.
- Once the core architecture was humming, we shifted my focus to **production-readiness**. We started swapping out the mock components. For web search, we wired up a real API. Redis became the go-to for distributed memory storage. Authentication and rate limiting followed soon after, essentials for a secure, multi-user environment.
- Scaling was next on our mind. We introduced **job queues** like Celery to manage long-running processes and added **websockets** for real-time updates. Eventually, we started thinking about **horizontal scaling**, building the infrastructure so that multiple nodes could work in harmony when load increased.
- Monitoring came hand-in-hand with scale. We added **comprehensive logging**, set up **metrics collection**, and built out **alerting systems** to get notified when

something broke. That gave us peace of mind and made debugging far less of a guessing game.

- And then, of course, there's **security**. Every input goes through validation and sanitization. API keys are managed securely, and **rate limiting** ensures no single user can abuse the system. These are the boring but critical things that keep everything running smoothly.
- Finally, we couldn't ignore **costs**. we implemented usage tracking, added **token-based quotas**, and gave users an estimate of how much their request would cost *before* running it. It's the kind of transparency you'd want.

And every one of these learnings came from seeing things break, learning from the experience, and building something better the next time around.

This implementation provides a solid foundation for a multi-agent research system while remaining simple enough to understand and extend.

The modular architecture makes it easy to swap components, add new tools, or modify agent behaviors as needed.

Hope you find the walk-through useful!

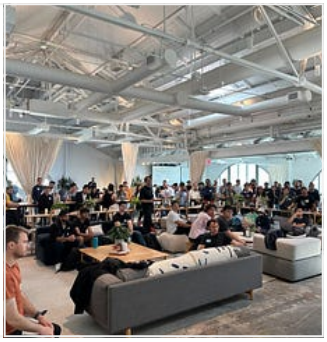
Bonus Content : Building with AI

And don't forget to have a look at some practitioner resources that we published recently:

Anthropic Teams' 100x Claude Workflows: What They Didn't Tell You, Until Now

The following is a battle-tested, developer-to-developer walkthrough on turning Claude into a genuine teammate rather...

agentissue.medium.com



The Ultimate Guide to OAuth2 on X: Secure Auth and Token Management for Social Scheduling

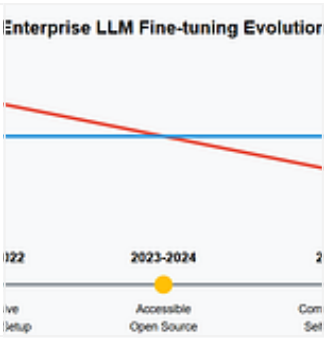
We recently added user account linking to our SaaS platform, which helps technical writers, researchers, and developers...

agentissue.medium.com

LLM Fine-Tuning Strategy: 4 Open Source Toolkits That You Should Know

With everything at our fingertips today, fine-tuning large language models (LLMs) can get overwhelming fast.

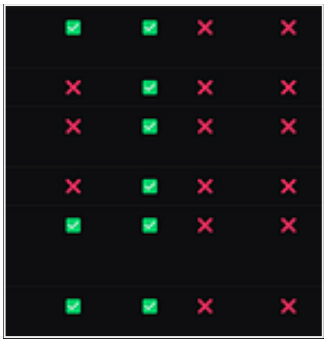
agentissue.medium.com



Building a Plug-and-Play Marketing Analytics Agent with the Model Context Protocol (MCP)

Over the past few months I've been experimenting with the Model Context Protocol.

agentissue.medium.com



Thank you for stopping by again, see you around.

Anthropic Claude | Multi Agent Systems | Agentic Workflow | Ai Agent Service | Ai Automation

Follow

Written by Agent Native

3.2K followers · 0 following

Your front-row seat to the future of Agents.

Responses (1)

Bgerby

What are your thoughts?

Thanawat Raibroycharoen
Jun 27

Thanks for sharing

Reply

More from Agent Native

Agent Native

LangChain and LangGraph v1.0: Beyond Release Notes, Into Real ROI

Today, I'm not here to recap release notes. I'll share what actually works across industries, and how it maps to the new features in v1.0.

4d ago 6

Agent Native

Anthropic Teams' 100x Claude Workflows: What They Didn't Tell You, Until Now

The following is a battle-tested, developer-to-developer walkthrough on turning Claude into a genuine teammate rather than "just another...

Jun 8 388 12

Agent Native

DeepSeek-OCR: Context Compressor for Enterprise Agents

I went into this release expecting “another OCR,” then had one of those brain-tilt moments where your mental model updates in real time.

2d ago 1

Agent Native

Optimizing Prompts for Language Model Pipelines: DSPy MIPROv2

I’ve been diving deep into prompt optimization for large language models (LLMs) lately, especially as we build more complex NLP pipelines —...

Oct 29, 2024 86

See all from Agent Native

Recommended from Medium

In Towards AI by Teja Kusireddy

We Spent $47,000 Running AI Agents in Production. Here’s What Nobody Tells You About A2A and MCP.

Multi-agent systems are the future. Agent-to-Agent (A2A) communication and Anthropic’s Model Context Protocol (MCP) are revolutionary. But...

Oct 16 1.6K 37

In Coding Nexus by Code Coup

Claude Desktop Might Be the Most Useful Free Tool You’ll Install This Year

I didn't expect much when I first saw the announcement for Claude Desktop. Another AI wrapper, I thought. Maybe with a shiny UI.

Oct 23 297 12

In Dare To Be Better by Max Petrusenko

Unleash Your Inner Wizard: Claude Skills Are Automating Enterprise for Pennies

How a simple folder is replacing $50K consultants and saving companies literal days of work

Oct 17 452 6

In Level Up Coding by Lak Lakshmanan

Comparing Agent Frameworks: PydanticAI, LangChain 1.0 and Google ADK

Is LangChain 1.0 ready for prime time? Is Google ADK worth betting on? Should we stick with Pydantic AI?

5d ago 28

In AI Software Engineer by Joe Njenga

Cursor 2.0 Has Arrived — And Agentic AI Coding Just Got Wild

Cursor has released version 2.0 , bringing the most powerful agentic AI we have seen yet, more autonomous than ever before,here's what's...

2d ago 222 5

In Data Science Collective by Ida Silfverskiöld

Agentic AI: Single vs Multi-Agent Systems

Building with a structured data source in LangGraph

3d ago 378 9