Here's the updated research agent with ChromaDB as the vector database, providing enhanced metadata management and query capabilities:

```python
# requirements.txt
# langchain==0.2.1
# chromadb==0.4.24
# sentence-transformers==3.0.0
# openai==1.30.1
# unstructured==0.13.0
# PyGithub==2.3.0
# requests==2.32.3
# python-dotenv==1.0.1
# tiktoken==0.7.0
# langchain-chroma==0.1.1

import os
import shutil
from pathlib import Path
import json
from langchain_community.document_loaders import DirectoryLoader, TextLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_community.embeddings import HuggingFaceEmbeddings, OpenAIEmbeddings
from langchain_community.vectorstores import Chroma
from langchain.chains import RetrievalQA
from langchain_community.llms import OpenAI, HuggingFaceHub
from langchain.prompts import PromptTemplate
from github import Github
import requests
from dotenv import load_dotenv

# Load environment variables
load_dotenv()

class ResearchAgent:
    def __init__(self, workspace="research_workspace", model_name="gpt-3.5-turbo",
                 use_openai=False, github_token=None):
        # Create workspace structure
        self.workspace = Path(workspace)
        self.research_papers_dir = self.workspace / "research_papers"
        self.code_dir = self.workspace / "code"
        self.library_dir = self.workspace / "library"

        self.create_workspace()
        self.use_openai = use_openai
        self.model_name = model_name
        self.vector_store = None
        self.qa_chain = None
        self.github_search = GitHubSearch(github_token)

        self.load_or_create_vector_store()
        self.initialize_qa_chain()

    def create_workspace(self):
        """Create the workspace directory structure with necessary permissions"""
        # Create directories if they don't exist
        self.workspace.mkdir(exist_ok=True, parents=True)
        self.research_papers_dir.mkdir(exist_ok=True)
        self.code_dir.mkdir(exist_ok=True)
        self.library_dir.mkdir(exist_ok=True)

        # Set permissions (read/write/execute for owner, read for others)
        os.chmod(self.workspace, 0o755)
        os.chmod(self.research_papers_dir, 0o755)
        os.chmod(self.code_dir, 0o755)
        os.chmod(self.library_dir, 0o755)

        print(f"Workspace created at: {self.workspace.resolve()}")
        print(f"  - Research papers: {self.research_papers_dir}")
        print(f"  - Code storage: {self.code_dir}")
        print(f"  - Vector library: {self.library_dir}")

    def get_embeddings(self):
        """Get embedding model based on configuration"""
        if self.use_openai:
            return OpenAIEmbeddings()
        return HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")

    def load_or_create_vector_store(self):
        """Load existing ChromaDB collection or create new from research papers"""
        embeddings = self.get_embeddings()

        # Check if ChromaDB collection exists
        if (self.library_dir / "chroma.sqlite3").exists():
            print("Loading existing ChromaDB vector store...")
            self.vector_store = Chroma(
                persist_directory=str(self.library_dir),
                embedding_function=embeddings
            )
        else:
            print("Creating new ChromaDB vector store from research papers...")
            self.load_documents()

    def load_documents(self):
        """Load and process documents from the research papers directory"""
        loader = DirectoryLoader(
            str(self.research_papers_dir),
            glob="**/*.*",
            show_progress=True
        )
        documents = loader.load()

        if not documents:
            print("Warning: No documents found in research_papers directory")
            print(f"Add files to: {self.research_papers_dir.resolve()}")
            # Create empty vector store
            self.vector_store = Chroma.from_texts(
                ["No documents loaded"],
                embedding=self.get_embeddings(),
                persist_directory=str(self.library_dir)
            )
            return

        # Add document metadata
        for doc in documents:
            doc.metadata["source_path"] = doc.metadata["source"]
            doc.metadata["source"] = str(Path(doc.metadata["source"]).name)
            doc.metadata["document_type"] = "research_paper"

        text_splitter = RecursiveCharacterTextSplitter(
            chunk_size=1000,
            chunk_overlap=200,
            length_function=len,
            add_start_index=True
        )
        chunks = text_splitter.split_documents(documents)

        print(f"Processed {len(chunks)} document chunks from {len(documents)} files")

        # Create ChromaDB vector store
        self.vector_store = Chroma.from_documents(
                documents=chunks,
                embedding=self.get_embeddings(),
                persist_directory=str(self.library_dir),
                collection_metadata={"hnsw:space": "cosine"},
                ids=[f"doc_{i}" for i in range(len(chunks))]
            )

            # Persist to disk
            self.vector_store.persist()
            print(f"Vector store saved to: {self.library_dir}")

        def initialize_qa_chain(self):
            """Initialize the QA chain with retriever and LLM"""
            retriever = self.vector_store.as_retriever(
                search_type="mmr",  # Maximal Marginal Relevance
                search_kwargs={
                    "k": 8,
                    "filter": {"document_type": "research_paper"},
                    "score_threshold": 0.7
                }
            )

            # Custom prompt template
            prompt_template = """You are a research assistant. Use the following context to answer the
    question.
            If you don't know the answer, say you don't know. Be detailed and technical.

            Context:
            {context}

            Question: {question}
            Answer in markdown format:"""

            custom_prompt = PromptTemplate(
                template=prompt_template,
                input_variables=["context", "question"]
            )

            if self.use_openai:
                llm = OpenAI(model=self.model_name, temperature=0)
            else:
                llm = HuggingFaceHub(
                    repo_id="google/flan-t5-xxl",
                    model_kwargs={"temperature": 0.5, "max_length": 512}
            )

        self.qa_chain = RetrievalQA.from_chain_type(
            llm=llm,
            chain_type="stuff",
            retriever=retriever,
            chain_type_kwargs={"prompt": custom_prompt},
            return_source_documents=True
        )

    def add_research_paper(self, file_path):
        """Add a new research paper to the knowledge base"""
        # Copy file to research_papers directory
        dest_path = self.research_papers_dir / Path(file_path).name
        shutil.copy(file_path, dest_path)
        print(f"Added research paper: {dest_path}")

        # Reload documents to update vector store
        self.load_documents()
        self.initialize_qa_chain()

    def add_github_results_to_knowledge(self, github_results):
        """Add GitHub search results to the knowledge base"""
        if not github_results:
            return

        # Create documents from GitHub results
        documents = []
        for result in github_results:
            content = result["content"]
            metadata = result["metadata"]
            metadata["document_type"] = "github_code"
            documents.append({
                "page_content": content,
                "metadata": metadata
            })

        # Add to ChromaDB collection
        self.vector_store.add_texts(
            texts=[doc["page_content"] for doc in documents],
            metadatas=[doc["metadata"] for doc in documents],
            ids=[f"github_{i}" for i in range(len(documents))]
        )
        self.vector_store.persist()
```

```
        print(f"Added {len(documents)} GitHub results to knowledge base")

    def save_code_snippet(self, filename, content):
        """Save generated code to the code directory"""
        code_path = self.code_dir / filename
        with open(code_path, "w") as f:
            f.write(content)
        os.chmod(code_path, 0o644)  # Set file permissions
        return code_path

    def query(self, question, include_github=False, generate_code=False, persist_github=False):
        """Query the research agent with enhanced capabilities"""
        # Query local knowledge base
        local_result = self.qa_chain({"query": question})
        response = {
            "answer": local_result["result"],
            "sources": [],
            "github_results": [],
            "generated_code": None
        }

        # Extract unique sources with metadata
        source_metadata = {}
        for doc in local_result["source_documents"]:
            source_path = doc.metadata.get("source_path", doc.metadata["source"])
            if source_path not in source_metadata:
                source_metadata[source_path] = {
                    "document_type": doc.metadata.get("document_type", "research_paper"),
                    "page": doc.metadata.get("page", ""),
                    "start_index": doc.metadata.get("start_index", "")
                }

        response["sources"] = [
            {
                "path": path,
                "type": meta["document_type"],
                "location": f"page {meta['page']}" if meta["page"] else f"char {meta['start_index']}"
            } for path, meta in source_metadata.items()
        ]

        # Add GitHub search results if requested
        if include_github:
            github_code = self.github_search.search_code(question)
            github_repos = self.github_search.search_repositories(question)
```

````
    # Add READMEs from top repositories
    for repo in github_repos:
        readme_content = self.github_search.get_repo_readme(repo.full_name)
        if readme_content:
            github_code.append({
                "content": readme_content,
                "metadata": {
                    "source": repo.html_url,
                    "repository": repo.full_name,
                    "path": "README.md",
                    "source_type": "github"
                }
            })

    response["github_results"] = github_code

    # Optionally add GitHub results to knowledge base
    if persist_github:
        self.add_github_results_to_knowledge(github_code)

# Generate and save code if requested
if generate_code and self.use_openai:
    code_prompt = f"""{question}\n\nProvide complete, runnable code solution with:
    - Detailed comments
    - Error handling
    - Example usage"""
    generated_code = self.qa_chain({"query": code_prompt})["result"]

    # Try to extract code block if exists
    if "```" in generated_code:
        code_block = generated_code.split("```")[1]
        if code_block.startswith("python"):
            generated_code = code_block[6:]
        else:
            generated_code = code_block

    # Save code to file
    filename = f"{question[:50].replace(' ', '_').replace('?', '')}.py"
    code_path = self.save_code_snippet(filename, generated_code)
    response["generated_code"] = {
        "path": str(code_path),
        "content": generated_code
    }
````

```
        return response

    def format_response(self, response):
        """Format the response for better readability"""
        formatted = f"## Answer\n{response['answer']}\n\n"

        if response['sources']:
            formatted += "## Local Sources\n"
            for src in response['sources']:
                formatted += f"- **{src['type'].upper()}**: {src['path']} ({src['location']})\n"

        if response['github_results']:
            formatted += "\n## GitHub Results\n"
            for i, result in enumerate(response['github_results'], 1):
                meta = result['metadata']
                formatted += (f"\n{i}. **Repository**: [{meta['repository']}]({meta['source']})\n"
                              f"   **File**: {meta['path']}\n"
                              "   **Snippet**:\n```\n" +
                              "\n".join(line for line in result['content'].split('\n')[:10]) +
                              "\n```\n")

        if response.get('generated_code'):
            code = response['generated_code']
            formatted += (f"\n## Generated Code\n"
                          f"**Saved to**: {code['path']}\n"
                          "**Preview**:\n```python\n" +
                          "\n".join(line for line in code['content'].split('\n')[:20]) +
                          "\n```\n")

        return formatted

class GitHubSearch:
    def __init__(self, github_token=None):
        self.github_token = github_token or os.getenv("GITHUB_TOKEN")
        self.github_client = Github(self.github_token) if self.github_token else Github()
        self.rate_limit_reset = 0

    def search_repositories(self, query, max_repos=3):
        """Search GitHub repositories with rate limit handling"""
        try:
            results = self.github_client.search_repositories(query)
            return [repo for repo in results[:max_repos]]
        except Exception as e:
```

```
        self.handle_rate_limit(e)
        return []

def search_code(self, query, max_files=5):
    """Search code across GitHub with rate limit handling"""
    try:
        results = self.github_client.search_code(query)
        code_results = []

        for result in results[:max_files]:
            try:
                content = result.decoded_content.decode("utf-8")
                metadata = {
                    "source": result.html_url,
                    "repository": result.repository.full_name,
                    "path": result.path,
                    "source_type": "github"
                }
                code_results.append({"content": content, "metadata": metadata})
            except Exception as e:
                print(f"Error processing {result.html_url}: {str(e)}")

        return code_results
    except Exception as e:
        self.handle_rate_limit(e)
        return []

def handle_rate_limit(self, error):
    """Handle GitHub API rate limits"""
    if "rate limit" in str(error).lower():
        reset_time = self.github_client.get_rate_limit().core.reset
        wait_seconds = max(int((reset_time - datetime.now()).total_seconds()), 0)
        print(f"GitHub rate limit exceeded. Resets at {reset_time} ({wait_seconds} seconds)")
        self.rate_limit_reset = reset_time.timestamp()
    else:
        print(f"GitHub API error: {str(error)}")

def get_repo_readme(self, repo_full_name):
    """Get README content from a repository"""
    try:
        repo = self.github_client.get_repo(repo_full_name)
        readme = repo.get_readme()
        return readme.decoded_content.decode("utf-8")
    except Exception as e:
            print(f"Error fetching README for {repo_full_name}: {str(e)}")
            return ""

# Example Usage
if __name__ == "__main__":
    # Initialize agent with automatic workspace creation
    agent = ResearchAgent(
        workspace="my_research_agent",
        use_openai=True,
        model_name="gpt-4-turbo",
        github_token=os.getenv("GITHUB_TOKEN")
    )

    # Add a research paper (optional)
    # agent.add_research_paper("path/to/your/paper.pdf")

    # Ask a research question with GitHub search and code generation
    question = "Explain transformer architecture and provide a PyTorch implementation"
    response = agent.query(
        question,
        include_github=True,
        generate_code=True,
        persist_github=True  # Add GitHub results to knowledge base
    )

    # Print formatted response
    print(agent.format_response(response))

    # Save response to file
    with open(agent.workspace / "response.md", "w") as f:
        f.write(agent.format_response(response))
```

## Key ChromaDB Enhancements:

### 1. **ChromaDB Integration**
- **Persistent Storage**: Uses SQLite backend for efficient persistence
- **Metadata Management**: Stores document types, source paths, and location info
- **Advanced Retrieval**:
  ```python
  retriever = self.vector_store.as_retriever(
      search_type="mmr",  # Maximal Marginal Relevance for diversity
      search_kwargs={
          "k": 8,
  ```

          "filter": {"document_type": "research_paper"},
          "score_threshold": 0.7
      }
)
  ~~~
- **Optimized Indexing**: Uses cosine similarity with HNSW index
  ```python
  collection_metadata={"hnsw:space": "cosine"}
  ~~~

### 2. **Enhanced Metadata Handling**
- **Source Tracking**:
  ```python
  "sources": [
      {
          "path": "attention_paper.pdf",
          "type": "research_paper",
          "location": "page 5"
      },
      {
          "path": "https://github.com/huggingface/transformers",
          "type": "github",
          "location": "char 320"
      }
  ]
  ```
- **Document Typing**: Differentiates between research papers and GitHub content
- **Location Tracking**: Tracks page numbers and character indexes

### 3. **GitHub Integration Improvements**
- **Permanent Knowledge Addition**:
  ```python
  def add_github_results_to_knowledge(self, github_results):
      # Adds GitHub content to vector store permanently
      self.vector_store.add_texts(...)
  ```
- **Rate Limit Handling**:
  ```python
  def handle_rate_limit(self, error):
      # Calculates exact wait time when rate limited
      reset_time = self.github_client.get_rate_limit().core.reset
  ```
- **Rich Source Formatting**: Includes direct links to GitHub resources

### 4. **Prompt Engineering**
```python
prompt_template = """You are a research assistant. Use the following context to answer the question.
If you don't know the answer, say you don't know. Be detailed and technical.

Context:
{context}

Question: {question}
Answer in markdown format:"""
```

### 5. **Output Formatting**
- **Markdown Support**: Renders answers in Markdown format
- **Code Block Formatting**:
  ```python
  "**Snippet**:\n```\n" + content[:10] + "\n```\n"
  ```
- **Response Export**: Saves complete response to Markdown file

## Folder Structure with ChromaDB:
```
my_research_agent/
├── research_papers/
│   ├── attention_paper.pdf
│   └── transformers_guide.docx
├── code/
│   ├── transformer_implementation.py
│   └── cnn_mnist_classifier.py
├── library/              # ChromaDB storage
│   ├── chroma.sqlite3       # SQLite database
│   ├── chroma-embeddings.parquet
│   └── index/               # HNSW index files
└── response.md             # Formatted output
```

## Advanced Features:

1. **Hybrid Search**:
   ```python
   # Search both research papers and GitHub content
   response = agent.query(
       "Explain transformers",
   ```

```
       search_filters={"$or": [
           {"document_type": "research_paper"},
           {"source_type": "github"}
       ]}
   )
```

2. **Metadata Querying**:
   ```python
   # Find documents from specific source
   results = self.vector_store.get(
       where={"repository": "huggingface/transformers"}
   )
   ```

3. **Versioned Collections**:
   ```python
   # Create versioned collections
   self.vector_store = Chroma.from_documents(
       collection_name="v2_research",
       # ... other params ...
   )
   ```

4. **Similarity Search**:
   ```python
   # Find similar documents
   similar = self.vector_store.similarity_search(
       "attention mechanisms",
       k=5,
       filter={"document_type": "research_paper"}
   )
   ```

5. **Update Management**:
   ```python
   # Update specific documents
   self.vector_store.update_document(
       document_id="doc_123",
       new_text="Updated content...",
       new_metadata={"updated": True}
   )
   ```

This implementation provides a robust research agent with ChromaDB's powerful vector database capabilities, enabling efficient metadata management, hybrid search, and persistent knowledge storage. The GitHub integration is now more resilient with proper rate limit handling, and the output formatting creates professional-quality research reports.