Enhanced Implementation Plan for Lisa-AI MVP

1. Overview

The **Lisa-AI Project** utilizes CrewAI RAG agents to process PDF, DOCX, and TXT documents, extract relevant data, and generate structured psychoeducational reports. This MVP focuses on core functionalities, ensuring that data from one student is isolated from another through session control, without incorporating security and encryption features.

---

2. Project Structure

Organize your project to ensure maintainability and scalability while focusing on core functionalities.

plaintext

```
lisa-ai/
├── README.md
├── .env                    # Environment variables
├── pyproject.toml
├── src/
│   ├── config/
│   │   ├── agents.yaml     # Define agents
│   │   └── tasks.yaml      # Define tasks
```

```
|   ├── crew.py             # Logic to configure and run the crew
|   ├── main.py             # Entry point for local testing
|   └── tools/
|       ├── pdf_tool.py     # PDF RAG tool
|       ├── docx_tool.py    # DOCX RAG tool
|       ├── txt_tool.py     # TXT RAG tool
|       └── utils.py        # Utilities for switching LLMs and session
control
├── templates/
|   └── report_template.json # Report structure
└── data/
    ├── sample.pdf
    ├── sample.docx
    └── sample.txt
```

Changes:

- Removed the `tests/` directory as testing is excluded from the initial MVP.
- Removed `output/` directory since report generation can directly output to a specified path.

---

3. Core Implementation Enhancements

3.1. Agents Configuration

Enhance agent configurations to include additional metadata and logging for better traceability.

config/agents.yaml:

yaml

```yaml
pdf_agent:
  role: PDF Processor
  goal: Extract and retrieve data from PDF files.
  tools:
    - pdf_tool
  backstory: >
    A skilled document analyzer for semantic searches in PDF files.
  metadata:
    author: "Lisa-AI Team"
    version: "1.0"
    last_updated: "2024-04-27"

docx_agent:
  role: DOCX Processor
  goal: Extract and retrieve structured data from DOCX files.
  tools:
```

```
    - docx_tool
  backstory: >
    A specialist in processing structured Word documents for key
insights.
  metadata:
    author: "Lisa-AI Team"
    version: "1.0"
    last_updated: "2024-04-27"


txt_agent:
  role: TXT Processor
  goal: Perform semantic searches in TXT files.
  tools:
    - txt_tool
  backstory: >
    Expert in processing unstructured text files for actionable
insights.
  metadata:
    author: "Lisa-AI Team"
    version: "1.0"
    last_updated: "2024-04-27"
```

Enhancements:

- **Metadata**: Adds authorship, versioning, and update timestamps for better maintenance.
- **Logging**: Implement logging within agents to capture processing statuses.

3.2. Tasks Configuration

Refine task descriptions to include dependencies and priority levels.

`config/tasks.yaml`:

yaml

```yaml
process_pdf_task:
  description: >
    Extract and retrieve contextually relevant data from PDF files.
  expected_output: >
    A structured summary of the content and semantic insights.
  priority: high
  dependencies: []
  timeline: "5 minutes"

process_docx_task:
  description: >
    Analyze and retrieve specific sections from DOCX files.
```

```
  expected_output: >
    Key insights indexed for report generation.
  priority: medium
  dependencies: [process_pdf_task]
  timeline: "3 minutes"


process_txt_task:
  description: >
    Perform semantic analysis and retrieval on TXT files.
  expected_output: >
    Relevant sections prepared for report generation.
  priority: medium
  dependencies: [process_pdf_task]
  timeline: "2 minutes"
```

Enhancements:

- **Priority Levels**: Assign priorities to manage task execution order.
- **Dependencies**: Define task dependencies to ensure correct processing sequences.
- **Timeline**: Estimate processing times to manage user expectations and system performance.

3.3. Tools Implementation

Simplify tools by removing encryption and security features, and implement session control to ensure data isolation between reports.

utils.py:

python

```
import os
import logging


def get_llm_config(provider="ollama"):
    """
    Return LLM configuration based on the selected provider.
    """
    if provider == "ollama":
        return {
            "llm": {
                "provider": "ollama",
                "config": {
                    "model": "llama2",
                    "temperature": 0.5
                }
            },
```

```
        "embedder": {
            "provider": "ollama",
            "config": {
                "model": "embedding-001"
            }
        }
    }
elif provider == "openai":
    return {
        "llm": {
            "provider": "openai",
            "config": {
                "model": "gpt-4",
                "temperature": 0.7
            }
        },
        "embedder": {
            "provider": "openai",
            "config": {
                "model": "text-embedding-ada-002"
            }
```

```
            }
        }
    else:
        raise ValueError("Unsupported LLM provider. Choose 'ollama' or
'openai'.")


def setup_logging():
    """
    Set up logging configuration.
    """
    logging.basicConfig(
        level=logging.INFO,
        format="%(asctime)s [%(levelname)s] %(message)s",
        handlers=[
            logging.FileHandler("lisa_ai.log"),
            logging.StreamHandler()
        ]
    )


def clear_session():
    """
```

```
    Clear any session-specific data after report generation.
    """
    logging.info("Clearing session data...")
    # Implement session data clearing logic if stored in memory or temporary files
    # For example, delete temporary files or reset in-memory data structures
    # Since data is processed per report, ensure no residual data remains
    pass

# Initialize logging at module load
setup_logging()
```

Enhancements:

- **Logging**: Comprehensive logging setup to track processing activities and errors.
- **Session Control**: Implement `clear_session` function to wipe out information after each report generation.

`pdf_tool.py`:

python

```
import logging
from crewai_tools import PDFSearchTool
from utils import get_llm_config

llm_config = get_llm_config(provider="ollama")  # Switch to "openai"
as needed

pdf_tool = PDFSearchTool(config=llm_config)

def search_pdf(file_path, query):
    try:
        logging.info(f"Processing PDF file: {file_path} with query:
{query}")
        tool = PDFSearchTool(pdf=file_path, config=llm_config)
        result = tool.run(query=query)
        logging.info(f"Successfully processed PDF file: {file_path}")
        return result
    except FileNotFoundError:
        logging.error(f"PDF file not found: {file_path}")
        return None
    except Exception as e:
```

```
        logging.error(f"Error processing PDF file {file_path}:
{str(e)}")
        return None
```

docx_tool.py:

python

```
import logging
from crewai_tools import DOCXSearchTool
from utils import get_llm_config

llm_config = get_llm_config(provider="ollama")  # Switch to "openai"
as needed

docx_tool = DOCXSearchTool(config=llm_config)

def search_docx(file_path, query):
    try:
        logging.info(f"Processing DOCX file: {file_path} with query:
{query}")
        tool = DOCXSearchTool(docx=file_path, config=llm_config)
```

```
        result = tool.run(query=query)
        logging.info(f"Successfully processed DOCX file: {file_path}")
        return result
    except FileNotFoundError:
        logging.error(f"DOCX file not found: {file_path}")
        return None
    except Exception as e:
        logging.error(f"Error processing DOCX file {file_path}:
{str(e)}")
        return None
```

txt_tool.py:

python

```
import logging
from crewai_tools import TXTSearchTool
from utils import get_llm_config

llm_config = get_llm_config(provider="ollama")  # Switch to "openai"
as needed
```

```
txt_tool = TXTSearchTool(config=llm_config)


def search_txt(file_path, query):
    try:
        logging.info(f"Processing TXT file: {file_path} with query:
{query}")
        tool = TXTSearchTool(txt=file_path, config=llm_config)
        result = tool.run(query=query)
        logging.info(f"Successfully processed TXT file: {file_path}")
        return result
    except FileNotFoundError:
        logging.error(f"TXT file not found: {file_path}")
        return None
    except Exception as e:
        logging.error(f"Error processing TXT file {file_path}:
{str(e)}")
        return None
```

Enhancements:

- **Removed Encryption**: All encryption-related code and dependencies have been removed.
- **Logging**: Enhanced logging to track processing steps and errors.

- **Session Control**: Session data is cleared after report generation (handled in `crew.py`).

3.4. Report Generation

Enhance the report generator to handle multiple sections and ensure data isolation per session.

`report_generator.py`:

python

```
import json
import logging
from utils import clear_session

def generate_report(data, template_path, output_path):
    """
    Generate a psychoeducational report based on retrieved data and template.
    """
    try:
        logging.info(f"Generating report using template: {template_path}")
        with open(template_path, "r") as template_file:
            template = json.load(template_file)
```

```
        report = {"title": template.get("title", "Psychoeducational
Report"), "sections": {}}

        for section, content in template.get("sections", {}).items():
            try:
                report["sections"][section] =
content.format(**data.get(section, {}))
            except KeyError as e:
                logging.error(f"Missing data for section '{section}':
{e}")
                report["sections"][section] = f"Error: Missing data
for {e}"

        with open(output_path, "w") as output_file:
            json.dump(report, output_file, indent=4)
        logging.info(f"Report successfully saved at {output_path}")

        # Clear session data after report generation
        clear_session()
        return True
    except FileNotFoundError:
```

```
            logging.error(f"Template file not found: {template_path}")
            return False
        except json.JSONDecodeError:
            logging.error(f"Invalid JSON format in template file:
{template_path}")
            return False
        except Exception as e:
            logging.error(f"Error generating report: {str(e)}")
            return False
```

Enhancements:

- **Session Control**: Calls `clear_session()` after report generation to ensure no residual data remains.
- **Data Validation**: Checks for missing data and logs errors, inserting error messages into the report where necessary.
- **Structured Formatting**: Uses Markdown-like formatting for readability.
- **Dynamic Title**: Allows for a default title if none is provided in the template.

`templates/report_template.json`: Ensure your report template is comprehensive and includes placeholders for dynamic data. Below is an enhanced example focusing on the "Reason for Referral" and "Background" sections.

json

```
{
```

```
    "title": "Psychoeducational Report",
    "sections": {
        "reason_for_referral": "### Reason for Referral\n\n**Referral Source:** {referral_source}\n\n**Primary Concerns:** {primary_concerns}\n\n**Parental Input:** {parental_input}\n\n**Teacher Feedback:** {teacher_feedback}\n",
        "background": "### Background Information\n\n**Family Information:** {family_information}\n\n**Language Proficiency:** {language_proficiency}\n\n**Social, Emotional, and Behavioral History:** {social_emotional_history}\n\n**Health and Developmental History:** {health_development}\n\n**Educational History:** {educational_history}\n"
    }
}
```

Enhancements:

- **Structured Formatting**: Uses Markdown-like formatting for readability.
- **Placeholders**: Clearly defined placeholders for dynamic data insertion.
- **Additional Sections**: Ready to include more sections (e.g., Cognitive Assessment, Recommendations) as the project scales.

3.5. Crew Logic

Enhance the crew logic to support session control and ensure data isolation between student reports.

crew.py:

python

```
import logging
from crewai import Agent, Task, Crew, Process
from src.tools.pdf_tool import search_pdf
from src.tools.docx_tool import search_docx
from src.tools.txt_tool import search_txt
from src.report_generator import generate_report
from src.utils import clear_session


def create_crew(provider="ollama"):
    """
    Create and configure the CrewAI crew with specified agents and
tasks.
    """
    # Configure agents with logging
    pdf_agent = Agent(role="PDF Processor", tools=[search_pdf],
metadata={"priority": "high"})
    docx_agent = Agent(role="DOCX Processor", tools=[search_docx],
metadata={"priority": "medium"})
```

```
    txt_agent = Agent(role="TXT Processor", tools=[search_txt],
metadata={"priority": "medium"})

    # Define tasks with dependencies
    pdf_task = Task(description="Process PDF files", agent=pdf_agent)
    docx_task = Task(description="Process DOCX files",
agent=docx_agent, dependencies=["process_pdf_task"])
    txt_task = Task(description="Process TXT files", agent=txt_agent,
dependencies=["process_pdf_task"])

    # Create crew with parallel processing
    crew = Crew(
        agents=[pdf_agent, docx_agent, txt_agent],
        tasks=[pdf_task, docx_task, txt_task],
        process=Process.parallel  # Enable parallel processing
    )
    return crew

def kickoff_crew(crew, inputs, template_path, output_path):
    """
    Start the crew with given inputs and generate the report.
```

```
    """
    try:
        logging.info("Starting CrewAI workflow...")
        results = crew.kickoff(inputs)

        # Structure data for report sections
        report_data = {
            "reason_for_referral": {
                "referral_source": results.get("pdf_agent", "N/A"),
                "primary_concerns": results.get("pdf_agent", "N/A"),
                "parental_input": results.get("docx_agent", "N/A"),
                "teacher_feedback": results.get("txt_agent", "N/A")
            },
            "background": {
                "family_information": results.get("pdf_agent", "N/A"),
                "language_proficiency": results.get("docx_agent",
"N/A"),
                "social_emotional_history": results.get("txt_agent",
"N/A"),
                "health_development": results.get("pdf_agent", "N/A"),
```

```
                "educational_history": results.get("docx_agent",
"N/A")
            }
        }

        # Generate report
        success = generate_report(report_data, template_path,
output_path)
        if success:
            logging.info("Report generation completed successfully.")
        else:
            logging.error("Report generation failed.")
    except Exception as e:
        logging.error(f"Error during CrewAI workflow: {str(e)}")
    finally:
        # Ensure session data is cleared even if an error occurs
        clear_session()
```

Enhancements:

- **Session Control**: Ensures that `clear_session()` is called in the `finally` block to wipe data regardless of success or failure.

- **Parallel Processing**: Enables simultaneous processing of multiple documents, improving efficiency.
- **Error Handling**: Comprehensive error handling to capture and log any issues during the workflow.

Note: Adjust the mapping of `results` to `report_data` based on actual extracted data structures.

3.6. Main File for Testing

Enhance the main file to accept dynamic inputs, handle errors, and integrate the report generator seamlessly.

`main.py`:

python

```
import logging
import argparse
import os
from src.crew import create_crew, kickoff_crew


def parse_arguments():
    """
    Parse command-line arguments for the application.
    """
```

```
    parser = argparse.ArgumentParser(description="Lisa-AI Report
Generator")
    parser.add_argument("--provider", type=str, choices=["ollama",
"openai"], default="ollama", help="LLM provider to use")
    parser.add_argument("--query", type=str, required=True,
help="Query to analyze student academic progress")
    parser.add_argument("--pdf", type=str, help="Path to PDF file")
    parser.add_argument("--docx", type=str, help="Path to DOCX file")
    parser.add_argument("--txt", type=str, help="Path to TXT file")
    parser.add_argument("--template", type=str,
default="templates/report_template.json", help="Path to report
template")
    parser.add_argument("--output", type=str,
default="output/report.json", help="Path to save the generated
report")
    return parser.parse_args()


def main():
    """
    Main function to run the Lisa-AI Report Generator.
    """
    # Set up logging
```

```
    logging.basicConfig(
        level=logging.INFO,
        format="%(asctime)s [%(levelname)s] %(message)s",
        handlers=[
            logging.FileHandler("lisa_ai.log"),
            logging.StreamHandler()
        ]
    )

    args = parse_arguments()

    # Validate input files
    for file_type in ['pdf', 'docx', 'txt']:
        file_path = getattr(args, file_type)
        if file_path and not os.path.isfile(file_path):
            logging.error(f"The specified {file_type.upper()} file
does not exist: {file_path}")
            return

    # Prepare inputs
    inputs = {
```

```
        "query": args.query,
        "pdf_file": args.pdf if args.pdf else "",
        "docx_file": args.docx if args.docx else "",
        "txt_file": args.txt if args.txt else ""
    }

    # Create and kickoff crew
    crew = create_crew(provider=args.provider)
    kickoff_crew(crew, inputs, args.template, args.output)
    logging.info(f"Report generated at {args.output}")

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

Enhancements:

- **Argument Parsing**: Uses `argparse` to allow dynamic input via command-line arguments.
- **Input Validation**: Checks if specified files exist before processing.
- **User Feedback**: Provides clear log messages indicating the status of operations.

Usage Example:

bash

```
python src/main.py --provider ollama --query "Analyze student academic progress" --pdf data/sample.pdf --docx data/sample.docx --txt data/sample.txt --template templates/report_template.json --output output/report.json
```

Enhancements:

- **Dynamic Inputs**: Users can specify different providers, queries, and file paths.
- **Error Handling**: Provides clear error messages if files are missing or paths are incorrect.

---

4. Deployment Enhancements

4.1. Local Deployment

Ensure that the application can run smoothly on the specified hardware (MacBook Pro Dual Core, Intel, 8 GB RAM).

Steps:

1. **Set Up Environment Variables**:

`.env`:

plaintext

```
OPENAI_API_KEY=your_openai_api_key
OLLAMA_API_KEY=your_ollama_api_key
```

-
- **Note**: Remove `ENCRYPTION_KEY` as encryption is no longer required.

Run the Application:

bash

```
python src/main.py --provider ollama --query "Analyze student academic progress" --pdf data/sample.pdf --docx data/sample.docx --txt data/sample.txt --template templates/report_template.json --output output/report.json
```

2.
3. **Monitoring Logs**:
    - Check `lisa_ai.log` for detailed processing logs and error messages.

4.2. CrewAI Enterprise Deployment

For scalability and collaboration, deploy via CrewAI+ with session control ensuring data isolation between student reports.

Steps:

Push to GitHub:

bash

```
git add .
git commit -m "Enhanced MVP implementation with session control and removed security features"
git push origin main
```

1.
2. **Deploy via CrewAI+**:
    - **Select Repository**: Choose your GitHub repository.
    - **Add Environment Variables**: Ensure `OPENAI_API_KEY` and `OLLAMA_API_KEY` are securely added to CrewAI+ environment variables.
    - **Configure Deployment Settings**: Set resource limits considering hardware constraints.
    - **Click Deploy**: Initiate deployment and monitor via CrewAI+ dashboard.

To implement the Lisa-AI project with support for PDF, DOCX, and TXT formats using the CrewAI RAG framework, we will integrate the PDFSearchTool, DOCXSearchTool, and TXTSearchTool. This system will allow efficient semantic searching, retrieval, and report generation from diverse file types. Here's the comprehensive implementation plan:

Plan Overview

1. Document Ingestion:
    - Automate parsing and indexing of PDF, DOCX, and TXT files.
    - Store embeddings in a FAISS vector store for retrieval.
2. Search Tools Integration:
    - Integrate PDFSearchTool, DOCXSearchTool, and TXTSearchTool for format-specific searches.

- Configure tools to use Ollama for local models (handling PII) and OpenAI for testing.

3. Multi-Agent System:
   - Design agents for file processing, semantic searching, and report generation.
4. Report Generation:
   - Create structured reports based on retrieved content.
5. Interactive User Queries:
   - Enable user queries on uploaded data via a chatbot agent.

Implementation Steps

1. Setup

Install Required Libraries:

```
pip install crewai[tools] faiss-cpu numpy pandas python-docx pdfminer.six
```

Environment Configuration: Set up environment variables for testing:

```
export OPENAI_API_KEY="your_openai_api_key"
```

Project Structure:

```
lisa-ai/ ├── config/ | ├── agents.yaml # Agent configurations | └── tasks.yaml # Task configurations ├── data/ | ├── raw/ # Raw input files | └── processed/ # Processed embeddings and metadata ├── embeddings/ | └── faiss_index/ # FAISS vector store ├── tools/ | ├── pdf_tool.py # PDFSearchTool integration | ├── docx_tool.py # DOCXSearchTool integration | ├── txt_tool.py # TXTSearchTool integration ├── main.py # Main pipeline orchestrator └── templates/ └── report_template.json # Report structure
```

---

2. File Parsing and Ingestion

Create `data_ingestion.py`:

```
from pdfminer.high_level import extract_text from docx import Document import os import json def parse_pdf(pdf_path): return extract_text(pdf_path) def parse_docx(docx_path): doc = Document(docx_path) return "\n".join([p.text for p in doc.paragraphs]) def parse_txt(txt_path): with open(txt_path, "r") as file: return file.read() def process_files(input_dir, output_path): data_chunks = [] for filename in os.listdir(input_dir): file_path = os.path.join(input_dir, filename) if filename.endswith(".pdf"): text = parse_pdf(file_path) elif filename.endswith(".docx"): text = parse_docx(file_path) elif filename.endswith(".txt"): text = parse_txt(file_path) else: continue data_chunks.append({"filename": filename, "content": text}) with open(output_path, "w") as output_file: json.dump(data_chunks, output_file) print(f"Processed {len(data_chunks)} files.")
```

---

3. Semantic Search Tools

PDF Search Integration (`pdf_tool.py`):

```
from crewai_tools import PDFSearchTool def search_pdf(pdf_path, query): tool = PDFSearchTool(pdf=pdf_path) return tool.run(query=query)
```

DOCX Search Integration (`docx_tool.py`):

```
from crewai_tools import DOCXSearchTool def search_docx(docx_path, query): tool = DOCXSearchTool(docx=docx_path) return tool.run(query=query)
```

TXT Search Integration (`txt_tool.py`):

```
from crewai_tools import TXTSearchTool def search_txt(txt_path, query): tool = TXTSearchTool(txt=txt_path) return tool.run(query=query)
```

---

4. Retrieval and Report Generation

Create Retrieval Module (`retrieval.py`):

```
import json from pdf_tool import search_pdf from docx_tool import search_docx from txt_tool import search_txt def retrieve_content(file_path, query): if file_path.endswith(".pdf"): return search_pdf(file_path, query) elif file_path.endswith(".docx"): return search_docx(file_path, query) elif file_path.endswith(".txt"): return search_txt(file_path, query) else: return f"Unsupported file format: {file_path}" def retrieve_from_directory(input_dir, query): results = [] for filename in os.listdir(input_dir): file_path = os.path.join(input_dir, filename) result = retrieve_content(file_path, query) results.append({"filename": filename, "result": result}) return results
```

Create Report Generator (`report_generator.py`):

```
import json def generate_report(retrieved_data, template_path, output_path): with open(template_path, "r") as f: template = json.load(f) report = {"title": template["title"], "sections": {}} for section, placeholder in template["sections"].items(): section_data = [item["result"] for item in retrieved_data if section in item["filename"]] report["sections"][section] =
```

```
"\n".join(section_data) with open(output_path, "w") as output_file:
json.dump(report, output_file, indent=4) print(f"Report generated at
{output_path}")
```

---

5. Orchestrating with CrewAI

Define Agents in `config/agents.yaml`:

```
retriever: role: File Search Agent goal: Retrieve content from PDF, DOCX, and TXT
files. tools: [pdf_tool.py, docx_tool.py, txt_tool.py] report_generator: role:
Report Generator goal: Generate a structured report from retrieved content. tools:
[report_generator.py] chatbot: role: User Interaction Agent goal: Answer user
queries about uploaded files. tools: [pdf_tool.py, docx_tool.py, txt_tool.py]
```

Define Tasks in `config/tasks.yaml`:

```
retrieve_data: description: Search for specific information in files.
expected_output: Retrieved content for the report. generate_report: description:
Use retrieved content to create a structured report. expected_output: A JSON
report formatted as per the template. chatbot_query: description: Answer user
queries about files in the directory. expected_output: Contextual answers based on
file content.
```

Orchestrate in `main.py`:

```
from crewai import Agent, Task, Crew, Process from retrieval import
retrieve_from_directory from report_generator import generate_report retriever =
Agent(role="Retriever", tools=["pdf_tool.py", "docx_tool.py", "txt_tool.py"])
generator = Agent(role="Report Generator", tools=["report_generator.py"])
```

```
retrieve_task = Task(description="Retrieve data from files.", agent=retriever)
generate_task = Task(description="Generate the final report.", agent=generator)
crew = Crew(agents=[retriever, generator], tasks=[retrieve_task, generate_task],
process=Process.sequential) inputs = { "query": "Describe the student's academic
progress", "template": "templates/report_template.json", "output_dir":
"data/raw/", } crew.kickoff(inputs)
```

---

6. Testing and Iteration

Run the pipeline with test data:

```
python main.py
```

Features and Benefits

1. Support for Multiple Formats: Handles PDF, DOCX, and TXT files.
2. Semantic Search: Integrated tools provide accurate results based on queries.
3. Multi-Agent Orchestration: Ensures modularity and clear task allocation.
4. Scalability: Easily extendable to support other file formats.
5. PII Compliance: Uses Ollama for local models to ensure data privacy.

This implementation ensures robust handling of diverse file formats for the Lisa-AI project while maintaining efficiency and accuracy. Let me know if you need further details or enhancements!