In-code documentation for the “Family Document Manager System"

The `Document` module is a critical part of the Family Document Manager System,designed to standardize the representation and management of document metadata.

By using Python's `dataclass` for simplicity and efficiency, this module enables

seamless serialization, storage, and retrieval of document information.

Key Features:

- Defines the core structure of a document with attributes such as title, description, classification, and file path.

- Provides utility methods for dynamic field retrieval and dictionary conversion, enhancing flexibility for integration with other components.

- Facilitates efficient interaction with repositories and user interfaces by providing a uniform data model.

Purpose:

The module simplifies document-related operations by encapsulating document properties and behaviors, making it easier for developers to implement and extend functionality without managing raw data manually.

Future extensions, such as validation methods or integration with advanced metadata standards, can further enhance the utility and robustness of the `Document` class.

Module: Document

This module defines the Document class, a core component of the Family Document Manager System. The Document class is used to model and manage the metadata and attributes of individual documents within the system.

Classes: Document: Represents a document and provides methods for accessing its metadata.

Dependencies: - dataclasses: Used for efficient class creation and serialization. - typing: Provides type hints for enhanced code readability and robustness. """

Import necessary modules for creating a robust document class

from dataclasses import dataclass, asdict from typing import List

@dataclass class Document: """ A class to represent a document in the Family Document Manager System.

```
Attributes:
    id (str): A unique identifier for the document.
    title (str): The name or title of the document.
    description (str): A brief description providing additional
context.
    classification (str): The classification or category of the
document.
    file_path (str): The file path where the document is stored.
    upload_date (str): The date and time when the document was
uploaded.
"""
id: str  # Unique identifier for the document
title: str  # Title or name of the document
description: str  # Description providing context about the document
classification: str  # Classification for organizational purposes
file_path: str  # Path to the physical or digital document
upload_date: str  # Timestamp of document creation or upload

@classmethod
def get_fields(cls) -> List[str]:
    """
    Returns a list of field names for the `Document` class.

    This method dynamically retrieves the names of all attributes in
```

```
the
    `Document` class. It is useful for tasks such as creating dynamic
forms
    or exporting metadata.

    Returns:
        List[str]: A list of attribute names as strings.
    """
    return list(cls.__annotations__.keys())  # Extract field names
from type annotations

def to_dict(self):
    """
    Converts the document instance into a dictionary.

    This method uses the `asdict` function from the `dataclasses`
module to
    convert the `Document` object into a dictionary for serialization
or storage.

    Returns:
        dict: A dictionary representation of the document.
    """
    return asdict(self)  # Serialize the document as a key-value
dictionary
```

TODO: Consider adding methods for field validation or transformation

e.g., ensure file paths are valid or titles conform to naming standards.

Module: GUI

This module provides a graphical user interface (GUI) for the Family Document Manager System. It allows users to interact with the system to add, update, delete, and search for documents.

Classes: DocumentManagementSystemGUI: Main GUI class for the document manager.

Dependencies: - tkinter: To build the graphical interface. - ttk: For enhanced widgets like tabs and buttons. - GUIDocumentHandler: Logic handler for document operations. """

```
import tkinter as tk from tkinter import ttk, messagebox from DocumentRepository
import DocumentRepository

class DocumentManagementSystemGUI: """ The main graphical user interface (GUI) for
the Family Document Manager System.

This class manages user interactions with the system, including
viewing,
adding, updating, and deleting documents.

Attributes:
    master (Tk): The root Tkinter window.
    repo (DocumentRepository): The document repository instance.
"""

def __init__(self, master):
    """
    Initializes the GUI.

    Args:
        master (Tk): The root window of the application.
    """
    self.master = master
    self.master.title("Family Document Manager System")
    self.repo = DocumentRepository()
    self.create_widgets()

def create_widgets(self):
    """
    Creates the main interface with tabs for different
functionalities.

    Tabs include:
        - Create Document
        - Update Document
        - Delete Document
        - Search Documents
    """
    notebook = ttk.Notebook(self.master)
    notebook.pack(expand=True, fill="both")

    self.create_tab = ttk.Frame(notebook)
    notebook.add(self.create_tab, text="Create Document")

    self.setup_create_tab()
```

```
def setup_create_tab(self):
    """
    Configures the 'Create Document' tab with input fields and a
button.
    """
    ttk.Label(self.create_tab, text="Title").grid(row=0, column=0)
    title_entry = ttk.Entry(self.create_tab)
    title_entry.grid(row=0, column=1)

    # TODO: Add validation for required fields before submission.
```

Module: GUI

This module provides a graphical user interface (GUI) for the Family Document Manager System. It allows users to interact with the system to add, update, delete, and search for documents.

Classes: DocumentManagementSystemGUI: Main GUI class for the document manager.

Dependencies: - tkinter: To build the graphical interface. - ttk: For enhanced widgets like tabs and buttons. - GUIDocumentHandler: Logic handler for document operations. """

import tkinter as tk from tkinter import ttk, messagebox from DocumentRepository import DocumentRepository

class DocumentManagementSystemGUI: """ The main graphical user interface (GUI) for the Family Document Manager System.

```
This class manages user interactions with the system, including
viewing,
adding, updating, and deleting documents.

Attributes:
    master (Tk): The root Tkinter window.
    repo (DocumentRepository): The document repository instance.
"""
```

```
def __init__(self, master):
    """
    Initializes the GUI.

    Args:
        master (Tk): The root window of the application.
    """
    self.master = master
    self.master.title("Family Document Manager System")
    self.repo = DocumentRepository()
    self.create_widgets()

def create_widgets(self):
    """
    Creates the main interface with tabs for different
functionalities.

    Tabs include:
        - Create Document
        - Update Document
        - Delete Document
        - Search Documents
    """
    notebook = ttk.Notebook(self.master)
    notebook.pack(expand=True, fill="both")

    self.create_tab = ttk.Frame(notebook)
    notebook.add(self.create_tab, text="Create Document")

    self.setup_create_tab()

def setup_create_tab(self):
    """
    Configures the 'Create Document' tab with input fields and a
button.
    """
    ttk.Label(self.create_tab, text="Title").grid(row=0, column=0)
    title_entry = ttk.Entry(self.create_tab)
    title_entry.grid(row=0, column=1)

    # TODO: Add validation for required fields before submission.
```

Module: DocumentRepository

This module provides the `DocumentRepository` class, which manages a collection of documents in a family document management system. It handles operations such as adding, updating, searching, and removing documents, as well as saving them to and loading them from a CSV file.

Classes: DocumentRepository: Manages document storage and retrieval.

Dependencies: - csv: To read and write document data to CSV files. - datetime: To handle timestamps for documents. - pathlib: To manage file paths. - Document: A custom class that defines the structure of a document. """

import csv from datetime import datetime from pathlib import Path from Document import Document

class DocumentRepository: """ A repository for managing documents in the system.

```
This class provides functionality to load, save, search, add, and
remove
documents, as well as to generate associated text files for each
document.

Attributes:
    csv_file (Path): Path to the CSV file storing document metadata.
    documents (list): A list of `Document` objects managed by the
repository.
"""

def __init__(self, csv_file='data/document.csv'):
```

```
    """
    Initializes the DocumentRepository instance.

    Args:
        csv_file (str): The path to the CSV file used for storing
document metadata.
    """
    self.documents = []  # Internal storage for documents
    self.csv_file = Path(csv_file)  # CSV file path
    self.load_documents()  # Load existing documents on
initialization

def load_documents(self):
    """
    Loads documents from the CSV file into memory.

    Reads the CSV file and converts each row into a `Document`
object,
    which is stored in the `documents` list. Skips rows with invalid
data.

    Raises:
        FileNotFoundError: If the CSV file does not exist.
    """
    if self.csv_file.exists():
        with self.csv_file.open('r', newline='') as file:
            reader = csv.DictReader(file)
            for row in reader:
                self._add_document_from_row(row)
    else:
        print(f"File not found: {self.csv_file}. Starting with an
empty repository.")  # GOTCHA: Missing file handling.

def save_documents(self):
    """
    Saves all documents to the CSV file.

    Writes the current `documents` list to the CSV file. Ensures the
file
    directory exists before writing.
    """
    self.csv_file.parent.mkdir(parents=True, exist_ok=True)
    with self.csv_file.open('w', newline='') as file:
        writer = csv.DictWriter(file,
fieldnames=Document.get_fields())
```

```
        writer.writeheader()
        for doc in self.documents:
            writer.writerow(doc.to_dict())

def add_document(self, document):
    """
    Adds a new document to the repository and saves it.

    Args:
        document (Document): The document to add.
    """
    self.documents.append(document)
    self.save_documents()
    self.create_document_txt_file(document)

def search_documents(self, keyword):
    """
    Searches for documents containing a keyword.

    Args:
        keyword (str): The keyword to search for in document fields.

    Returns:
        list: A list of matching `Document` objects.
    """
    keyword = keyword.lower()
    return [doc for doc in self.documents if keyword in
str(doc.to_dict()).lower()]

def remove_document(self, document_id):
    """
    Removes a document and its associated files from the repository.

    Args:
        document_id (str): The unique ID of the document to remove.

    Returns:
        bool: True if the document was removed, False otherwise.

    TODO: Add error handling for cases where file deletion fails.
    """
    document = self.get_document_by_id(document_id)
    if document:
        # Delete associated files
        txt_file_path = self._generate_txt_file_path(document.title)
```

```
        if txt_file_path.exists():
            txt_file_path.unlink()
        # Remove document from memory
        self.documents = [doc for doc in self.documents if doc.id !=
document_id]
        self.save_documents()
        return True
    else:
        print(f"No document found with ID {document_id}.")
        return False
```

Module: DocumentRepository

This module provides the `DocumentRepository` class, which manages a collection of documents in a family document management system. It handles operations such as adding, updating, searching, and removing documents, as well as saving them to and loading them from a CSV file.

Classes: DocumentRepository: Manages document storage and retrieval.

Dependencies: - csv: To read and write document data to CSV files. - datetime: To handle timestamps for documents. - pathlib: To manage file paths. - Document: A custom class that defines the structure of a document. """

import csv from datetime import datetime from pathlib import Path from Document import Document

class DocumentRepository: """ A repository for managing documents in the system.

```
This class provides functionality to load, save, search, add, and
remove
documents, as well as to generate associated text files for each
document.

Attributes:
    csv_file (Path): Path to the CSV file storing document metadata.
    documents (list): A list of `Document` objects managed by the
repository.
"""

def __init__(self, csv_file='data/document.csv'):
```

```
    """
    Initializes the DocumentRepository instance.

    Args:
        csv_file (str): The path to the CSV file used for storing
document metadata.
    """
    self.documents = []  # Internal storage for documents
    self.csv_file = Path(csv_file)  # CSV file path
    self.load_documents()  # Load existing documents on
initialization

def load_documents(self):
    """
    Loads documents from the CSV file into memory.

    Reads the CSV file and converts each row into a `Document`
object,
    which is stored in the `documents` list. Skips rows with invalid
data.

    Raises:
        FileNotFoundError: If the CSV file does not exist.
    """
    if self.csv_file.exists():
        with self.csv_file.open('r', newline='') as file:
            reader = csv.DictReader(file)
            for row in reader:
                self._add_document_from_row(row)
    else:
        print(f"File not found: {self.csv_file}. Starting with an
empty repository.")  # GOTCHA: Missing file handling.

def save_documents(self):
    """
    Saves all documents to the CSV file.

    Writes the current `documents` list to the CSV file. Ensures the
file
    directory exists before writing.
    """
    self.csv_file.parent.mkdir(parents=True, exist_ok=True)
    with self.csv_file.open('w', newline='') as file:
        writer = csv.DictWriter(file,
fieldnames=Document.get_fields())
```

```
        writer.writeheader()
        for doc in self.documents:
            writer.writerow(doc.to_dict())

def add_document(self, document):
    """
    Adds a new document to the repository and saves it.

    Args:
        document (Document): The document to add.
    """
    self.documents.append(document)
    self.save_documents()
    self.create_document_txt_file(document)

def search_documents(self, keyword):
    """
    Searches for documents containing a keyword.

    Args:
        keyword (str): The keyword to search for in document fields.

    Returns:
        list: A list of matching `Document` objects.
    """
    keyword = keyword.lower()
    return [doc for doc in self.documents if keyword in
str(doc.to_dict()).lower()]

def remove_document(self, document_id):
    """
    Removes a document and its associated files from the repository.

    Args:
        document_id (str): The unique ID of the document to remove.

    Returns:
        bool: True if the document was removed, False otherwise.

    TODO: Add error handling for cases where file deletion fails.
    """
    document = self.get_document_by_id(document_id)
    if document:
        # Delete associated files
        txt_file_path = self._generate_txt_file_path(document.title)
```

```
        if txt_file_path.exists():
            txt_file_path.unlink()
        # Remove document from memory
        self.documents = [doc for doc in self.documents if doc.id !=
document_id]
        self.save_documents()
        return True
    else:
        print(f"No document found with ID {document_id}.")
        return False
```

Conclusion:

The `Document` module plays a foundational role in the Family Document Manager System

by ensuring consistency, reliability, and ease of interaction with document data.

Through a clear and extensible structure, it enables seamless integration across

other modules such as repositories, GUIs, and search utilities.

Future Work:

- Validation mechanisms can be added to ensure data integrity, such as verifying file paths or standardizing classification names.

- Methods for transforming or enriching document metadata (e.g., tagging or indexing) could further enhance functionality.

- Advanced error handling and logging capabilities can improve maintainability and user feedback.

By maintaining this modular and extensible approach, the `Document` module supports the system's long-term scalability and adaptability, making it a robust solution for managing family-related documents efficiently.

Developer's Guide: Family Document Manager System (FDMS)

Table of Contents

Overview

The **Family Document Manager System (FDMS)** is designed to streamline the organization, management, and retrieval of family-related documents. It features a graphical user interface (GUI) for user interactions and robust backend logic for document storage and search functionality.

Key Features:

- Add, update, delete, and search for documents.
- Classification and metadata management for each document.
- Persistent storage using CSV files with file handling for associated document text files.

For user setup and operation, refer to the fdms-ReadMe-PDF.pdf.

System Architecture

FDMS is a modular system with the following components:

- **GUI Module (`GUI.py`)**: Handles user interactions and interface logic using the Tkinter library.
- **Document Module (`document.py`)**: Defines the `Document` class for consistent document representation.
- **Document Repository (`DocumentRepository.py`)**: Manages storage, retrieval, and search operations for document metadata.
- **Data Storage**: CSV file-based storage system for metadata and text files for document details.

Key Architectural Principles:

- **Modularity**: Code is separated into logical components for maintainability.
- **Data Persistence**: Metadata stored in CSV format; files organized in the `data/` directory.
- **Extensibility**: The architecture allows for future integration of additional features like cloud storage.

Implementation Status

The following features from the original specifications (Hosawi-project Spec Ver 2.0) have been implemented:

1. **Document Management**:
    a. Adding new documents with metadata.
    b. Updating document metadata.
    c. Deleting documents and associated files.
2. **Search Functionality**:
    a. Keyword-based search across all fields.
3. **GUI Interaction**:
    a. Tabbed interface for Create, Update, Delete, and Search functionalities.
    b. Interactive forms for document input.

Not Yet Implemented:

- Advanced search filters (e.g., date range, classification-specific).
- User authentication or role management.
- Integration with external storage or APIs.

Setup and Deployment

Developer-Specific Notes:

- The project uses Python 3. Ensure that the correct version is installed on your machine.
- Install dependencies: `pip install -r requirements.txt`

- Directory Structure:
  - `data/`: Contains the `document.csv` file and document text files.
  - `DocumentRepository.py`, `document-py.py`, and `GUI.py`: Core modules.

Admin Tips:

- To reset the document repository, delete the `data/document.csv` file and associated text files. Restart the application to initialize a new repository.
- Test the system after any changes using the provided test cases.

Code Walkthrough

User Interaction Flow

The primary user interaction involves the following steps:

1. **Opening the Application**: The user launches the GUI, which initializes the repository and GUI handler.
2. **Choosing a Tab**: The user selects a tab for specific actions (Create, Update, Delete, or Search).
3. **Performing an Action**:
    a. **Create**: Inputs document details and saves them.
    b. **Update**: Selects an existing document, modifies details, and saves updates.
    c. **Delete**: Deletes a selected document and its files.
    d. **Search**: Searches for documents by keywords.

Flow in Code:

- **GUI.py**:
    - User inputs are collected via forms in `setup_create_tab`, `setup_update_tab`, etc.
    - These inputs are passed to the `GUIDocumentHandler` class for processing.
- **DocumentRepository.py**:
    - Actions like adding, updating, and deleting documents are executed and saved to the repository.
- **document-py.py**:
    - The `Document` class provides data models and utility methods for metadata handling.

Code Structure and Key Modules

GUI.py

- **Classes**:
    - `DocumentManagementSystemGUI`: Handles GUI interactions.
- **Methods**:

- `setup_create_tab`: Configures the "Create Document" tab.
- `setup_update_tab`: Handles the "Update Document" tab.
- `create_document`: Calls the repository to save a new document.

DocumentRepository.py

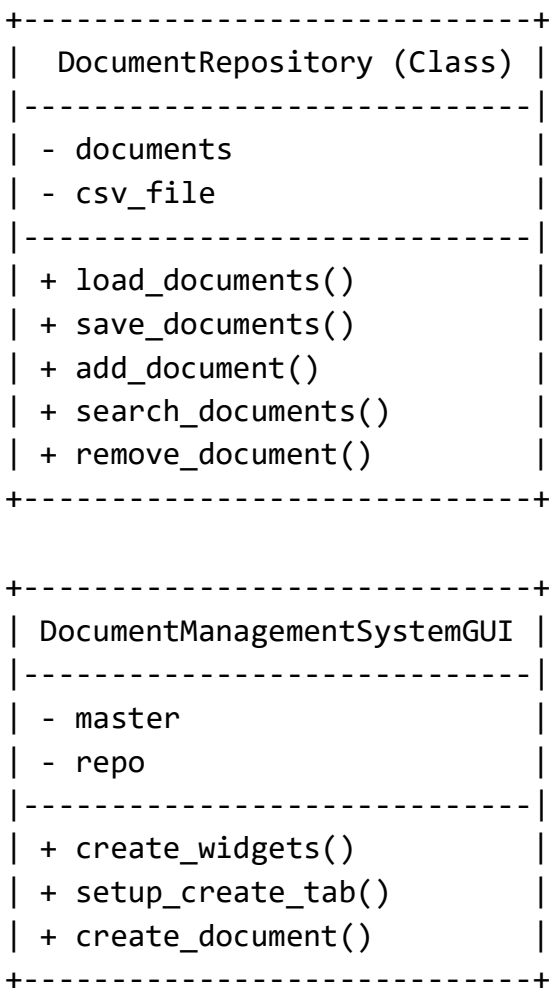
- **Classes**:
  - `DocumentRepository`: Manages document storage and retrieval.
- **Methods**:
  - `load_documents`: Loads data from the CSV file.
  - `add_document`: Adds a new document and creates associated text files.
  - `search_documents`: Searches for documents based on keywords.

document-py.py

- **Classes**:
  - `Document`: Defines document attributes and serialization methods.
- **Methods**:
  - `to_dict`: Converts a `Document` instance to a dictionary for storage.

Class Diagram

```
+-----------------------------+
|       Document (Class)      |
|-----------------------------|
| - id                        |
| - title                     |
| - description               |
| - classification            |
|                             |
                              |
|-----------------------------|
| + get_fields()              |
| + to_dict()                 |
+-----------------------------+
```


DocumentRepository (Class)
- documents
- csv_file
+ load_documents()
+ save_documents()
+ add_document()
+ search_documents()
+ remove_document()
DocumentManagementSystemGUI
- master
- repo
+ create_widgets()
+ setup_create_tab()
+ create_document()

Development Notes

1. **Error Handling**:
    a. Missing CSV files are logged, and the system initializes with an empty repository.
    b. TODO: Implement better exception handling for file operations.
2. **Validation**:
    a. Current validation is minimal. TODO: Add input validation for document fields.
3. **Testing**:
    a. Manual testing is currently required. Automate with a testing framework like `unittest`.

Future Enhancements

- **Search Filters**: Add advanced filtering by classification or date range.
- **File Attachments**: Allow upload and management of PDFs or images.
- **Multi-User Support**: Implement user authentication and role-based access.

This guide provides an overview and in-depth insights for developers to maintain and enhance the Family Document Manager System effectively. For basic setup, see thefdms-ReadMe-PDF.pdf.

Known Issues

Minor Issues

1. **Validation Gaps**:
    a. **Description**: The system currently does not enforce rigorous validation for user input fields in the GUI (e.g., empty or invalid entries for titles or descriptions).
    b. **Impact**: Could lead to inconsistent or incomplete data in the repository.
    c. **Fix**: Add validation checks in the GUI before saving data, ensuring all required fields are filled and properly formatted.
2. **UI Usability**:
    a. **Description**: Some button placements and labels may not be intuitive for non-technical users.
    b. **Impact**: Minor inconvenience or slower user adoption.
    c. **Fix**: Perform a UI/UX audit to improve layout and labeling for better usability.

Major Issues

1. **Scalability of CSV File Storage**:
    a. **Description**: The system relies on a single CSV file to store metadata. As the number of documents grows, loading and saving operations may become slower.
    b. **Impact**: Decreased performance with larger datasets, potentially leading to crashes.
    c. **Workaround**: Divide documents into multiple CSV files or implement lazy loading to minimize memory usage.
    d. **Fix**: Transition to a relational database (e.g., SQLite or PostgreSQL) to handle larger datasets efficiently.
2. **Concurrent Access**:
    a. **Description**: The system does not support concurrent access to the CSV file. Simultaneous reads/writes could corrupt the file.
    b. **Impact**: Severe data integrity issues in multi-user environments.

c. **Workaround**: Avoid running multiple instances of the application simultaneously.
d. **Fix**: Implement file-locking mechanisms or migrate to a database system with built-in concurrency handling.

Optional: Computational Inefficiencies

1. **Search Implementation**:
    a. **Description**: The keyword-based search scans all documents sequentially.
    b. **Impact**: Slow performance with larger datasets.
    c. **Fix**: Replace the linear search with an indexed search mechanism (e.g., binary search or database queries).
2. **File Handling**:
    a. **Description**: Creating and managing individual text files for each document can become inefficient with a large number of files.
    b. **Impact**: Increased disk I/O and potential file system clutter.
    c. **Fix**: Store document details in a single database or archive format rather than individual files.

Future Work

Expansion of Features

1. **Advanced Search Filters**:
    a. Add filtering options by date, classification, or other metadata attributes.
    b. Implement fuzzy matching for improved keyword search results.
2. **Multi-User Support**:
    a. Introduce user authentication and role-based access controls.
    b. Allow simultaneous access for multiple users with conflict resolution.
3. **Cloud Integration**:
    a. Provide options to store documents and metadata in cloud storage solutions like AWS S3 or Google Drive.
    b. Enable synchronization across devices for added convenience.
4. **Enhanced Metadata**:
    a. Allow tagging and keyword assignment for documents to improve searchability.
    b. Incorporate metadata standards for interoperability with other systems.

Addressing Inefficiencies

1. **Storage Migration**:
    a. Replace the current CSV-based storage with a lightweight relational database (e.g., SQLite).
    b. For scalability, consider NoSQL databases like MongoDB if hierarchical or non-tabular data is added.
2. **Search Optimization**:
    a. Build an index on document metadata for faster retrieval.
    b. Use full-text search libraries (e.g., Elasticsearch) for large datasets.
3. **Performance Enhancements**:
    a. Leverage multi-threading for tasks like file I/O and search operations.
    b. Optimize the GUI responsiveness by moving intensive operations to background threads.

Ongoing Deployment and Development

Ensuring Future Compatibility

1. **Unit Testing**:
    a. Implement a comprehensive test suite to cover critical operations like adding, deleting, and searching for documents.
    b. Automate testing with tools like `pytest` to ensure stability after updates.
2. **Documentation Updates**:
    a. Maintain updated user and developer documentation to reflect any changes or new features.
    b. Use a version-controlled repository (e.g., GitHub) for collaborative updates to documentation.
3. **Extensibility Guidelines**:
    a. New features should adhere to the modular architecture. For example:
        i. Use inheritance or composition to extend the `Document` class for additional fields or behavior.
        ii. Add new tabs or forms in the GUI following the existing pattern.
    b. Ensure backward compatibility when introducing major changes.
4. **Supporting New Formats**:

    a. Plan for supporting additional file types or metadata standards (e.g., PDFs, XML, JSON).
    b. Introduce a format-handling module to manage import/export operations.
5. **Regular Maintenance**:
    a. Periodically clean up unused files or metadata to prevent file system clutter.
    b. Monitor and optimize application performance, particularly in production environments.

Summary

This document outlines known issues, future development opportunities, and ongoing maintenance practices for the Family Document Manager System. By addressing scalability, performance, and usability, the system can evolve into a robust tool capable of managing large datasets and supporting diverse user needs. Regular updates and adherence to modular design principles will ensure that FDMS remains reliable and adaptable in the long term.