A REPORT ON THE

DESIGN AND IMPLEMENTATION OF A FILE ORGANIZER PROGRAM USING PYTHON

FOR THE 2025/2026 SESSION

BY

GROUP 7

MECHATRONICS ENGINEERING

SUBMITTED TO

MR AYUBA MUHAMMED

NEW HORIZONS- BELLS UNIVERSITY OF TECHNOLOGY

FEBRUARY 2026.

DESIGN AND IMPLEMENTATION OF A FILE ORGANIZER PROGRAM USING PYTHON

ABSTRACT

The rapid advancement of digital technology has significantly increased the volume of electronic files generated by individuals and organizations. These files include documents, images, videos, audio files, and software packages, which are often stored without proper structure. Poor file organization leads to inefficiency, wasted time, difficulty in retrieval, and an increased risk of data loss. Manual file organization is not only time-consuming but also inconsistent, especially as the number of files grows.

This project focuses on the design and implementation of an automated File Organizer Program using Python. The system automatically categorizes files in a selected directory based on their file extensions and organizes them into appropriate folders. The application is implemented as a desktop-based solution with a graphical user interface (GUI) to ensure ease of use for both technical and non-technical users. Python was selected due to its simplicity, cross-platform nature, and strong support for file handling and automation.

The developed system was tested using directories containing large numbers of mixed file types. The results demonstrate that the application successfully organizes files efficiently, reduces clutter, and improves accessibility. The project highlights the effectiveness of Python for desktop automation and provides a foundation for future improvements such as intelligent file classification and cloud integration.

GROUP 7 MEMBERS

NAME	MATRIC NUMBER	SIGNATURE
EGWUONWU PRAISE C.	2023/12408	
EBERECHI SAMUEL CHBUIKE	2023/12552	
EKE GOODNEWS	2023/12389	
EKEWUBA JESSE	2023/12303	
EKWUYASI EBUBE	2023/12427	

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 Background of the Study

The modern computing environment is characterized by the continuous creation and storage of digital data. Students, professionals, and organizations regularly generate documents, images, audio files, videos, and software installers during daily activities. As storage capacity becomes cheaper and more accessible, users tend to save files without proper organization. Over time, this practice results in cluttered directories that are difficult to manage and navigate.

File organization is an essential aspect of effective information management. A poorly organized file system can significantly reduce productivity, as users spend excessive time searching for files. In extreme cases, important files may be lost or duplicated unnecessarily. Automation has emerged as a reliable solution to address this challenge by reducing human involvement and ensuring consistency.

Python is a high-level programming language widely used for automation tasks due to its readability, flexibility, and extensive standard library. The availability of file-handling modules and GUI frameworks makes Python suitable for developing desktop applications that interact directly with the operating system. This project leverages these strengths to develop a File Organizer Program that simplifies file management.

1.2 Statement of the Problem

Many users store files in default folders such as Desktop and Downloads without categorization. As the number of files increases, it becomes difficult to locate specific items quickly. Manual organization requires constant effort and discipline, which most users fail to maintain consistently.

Although several file organization tools exist, many of them are either expensive, platform-specific, or require advanced configuration. Some applications are too complex for beginner users, while others lack flexibility. Therefore, there is a need for a simple, lightweight, and user-friendly file organizer that can operate offline and automate the organization process efficiently.

1.3 Aim and Objectives of the Study

The main aim of this project is to design and implement an automated File Organizer Program with GUI, and visualization using Python.

The specific objectives are:

1. To design a simple and intuitive graphical user interface
2. To scan files in a selected directory automatically
3. To categorize files based on file extensions
4. To create appropriate folders actively
5. To move files into their respective folders efficiently
6. To deploy on GitHub with complete documentation.

1.4 Significance of the Study

This project improves productivity by reducing the time and effort required to organize files. It also demonstrates the practical application of Python in solving real-world problems. The system serves as a learning resource for students interested in automation, file handling, and GUI development.

1.5 Scope and Limitations

The scope of this project is limited to organizing files within a user-selected local directory. The system categorizes files based on predefined file extensions. It does not analyse file content or integrate cloud storage services. The system relies solely on file extensions, which may not always accurately represent file content. It does not include undo functionality or advanced customization features. Performance may vary depending on system specifications.

1.6 Organization of the Report

This report is organized into five chapters covering the introduction, literature review, methodology, system implementation and results, and conclusion with recommendations.

CHAPTER 2: LITERATURE REVIEW

2.1 Analysis of Existing Related Applications

The problem of file clutter is already covered by a number of applications. These are divided into desktop-based and web-based solutions:

a) **Desktop Applications (DropIt & Hazel):** These are the windows and MacOS industry standards. They enable the user to write rules (when the file is a picture, transfer to Photos). Nevertheless, they tend to have a heavy background and may be tricky to configure for a layman user.

b) **Web-Based Organizers (Google drive/OneDrive Automation)**: Now cloud services have the feature of online organizing using folders through their Rules. They are powerful, but they need an active internet connection, and they are not useful with the local folder of the user, such as the Downloads and Desktop folders.

c) **OS Defaults:** Windows storage sense and macOS stacks provide an elementary type of grouping, but do not allow placement of files into certain and user-created category folders such as archives or university project.

2.2 Strengths and Weaknesses of Existing Tools

Tool	**Strengths**	**Weaknessses**
DropIt (Desktop)	Very customizable; lots of rules possible	Very steep learning curve, old-fashioned user interface
Hazel(macOS)	Strong integration with the OS; incredibly fast.	Premium (Paid license); Mac-only (No Windows).
Cloud Automators	Very accessible, anywhere you go	Needs internet; privacy issues (data on cloud).
Manual Sorting	100 percent control on each file.	Time-intensive; it is subject to human mistakes and oversights.

Table 2.2: Showing the strengths and weaknesses of existing tools.

2.3 <u>Python Libraries Used in Related Projects</u>

In order to automate to a professional level, a study indicates that the majority of Python-based organizers apply the following technology stack:

1. **OS and Shutil**: The fundamental libraries to navigate directories and the literal movement and copying of files.
2. **Watchdog**: A general library used in other such projects to give real time monitoring of folders.
3. **CustomTkinter (Our Choice):** In contrast to the standard Tkinter of simple student projects, CustomTkinter gives a high-definition (DPI-aware) interface and Dark Mode support. This is the typical flaw of Python apps where it appears to be old or ugly.
4. **Matplotlib/Pandas:** This is the version of this project used in data-intensive versions to visualize file distribution through charts.

2.4 <u>Gap Your Project Fills</u>

Although the above-mentioned tools are available, there is a Gap that the mentioned project intends to cover:

1. **Complexity Gap:** The majority of currently existing tools, such as DropIt, are too complex to be used by non-technical students. We offer a One-Click solution in our project.
2. **Aesthetic Gap**: The default free Python organizers are built using the old Tkinter that resembles Windows 95. With CustomTkinter, our project offers a modern UX (User Experience) in line with the existing Windows 11/macOS fashions.
3. **Offline & Privacy Gap:** This application is not web-based as the web-based tools are. No information is forwarded to the server and user privacy of sensitive documents is guaranteed.
4. **Reporting Gap:** This is because most of the organizers transfer files, which they are not telling you anything about. Our project has Matplotlib that will allow a user to have a visual report (Pie Chart) regarding his/her storage usage.

CHAPTER 3: METHODOLOGY

3.1 System Architecture

The system architecture consists of three major layers: the presentation layer, the application logic layer, and the file system layer. The presentation layer is responsible for user interaction through the graphical interface. The application logic layer processes user input and determines how files should be categorized. The file system layer performs the actual file operations.

This layered architecture improves maintainability and allows individual components to be modified without affecting the entire system. It also supports future expansion, such as adding new classification rules.

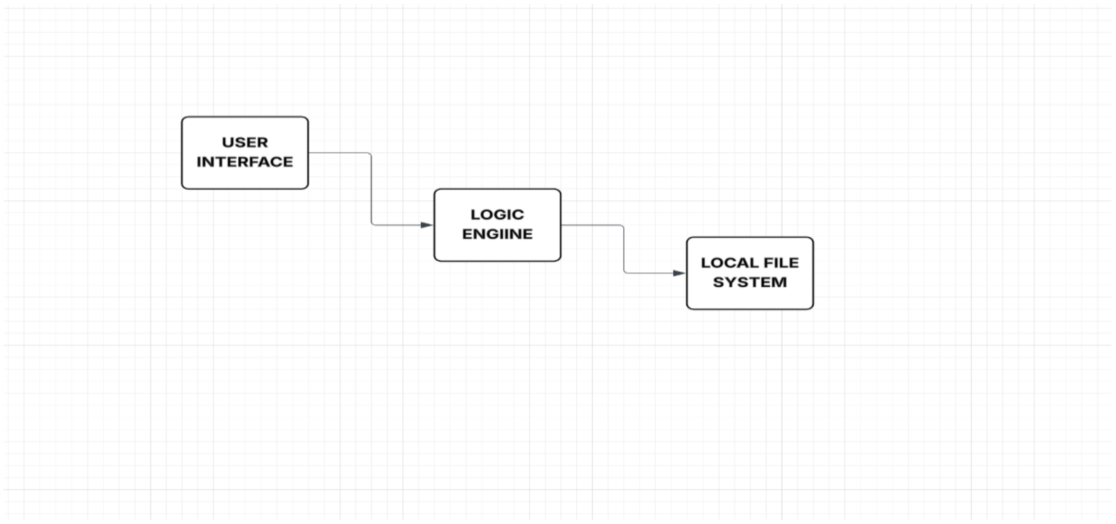


Figure 3.1 System Architecture

3.2 Tools & Libraries Used

Tool/Library	Version (Approx)	Justification
Python	3.10+	Primary language: chosen for its robust OS and shutil libraries.
Customtkinter	5.2.0	Provides a modern, dark-mode compatible GUI that looks professional.
Matplotlib	3.7.0	Used to generate interactive Pie Charts for storage visualization.
Watchdog	3.0.0	Enables real-time folder monitoring (the "Auto" part

		of the organizer).
JSON	Built-in	Used for config.json to remember user themes and recent folders.

Table 3.1 Tools and Libraries Used

3.3 GUI Design (Wireframe/Sketch):

The user interface was designed following a **Dashboard Layout** pattern. As shown in the wireframe below (Figure 3.2), the design prioritizes a clean user experience (UX) with the following components:

Sidebar (Left): Contains the primary navigation links (Organize, Search, Statistics, Tools, Settings).

Main Content Area (Center): Dynamically changes based on the user's selection to show folder paths or data charts.

Control Header: Provides quick access to the "Open Folder" and "Undo" actions.

Visual Feedback: Dedicated space at the bottom for status updates and progress logs.

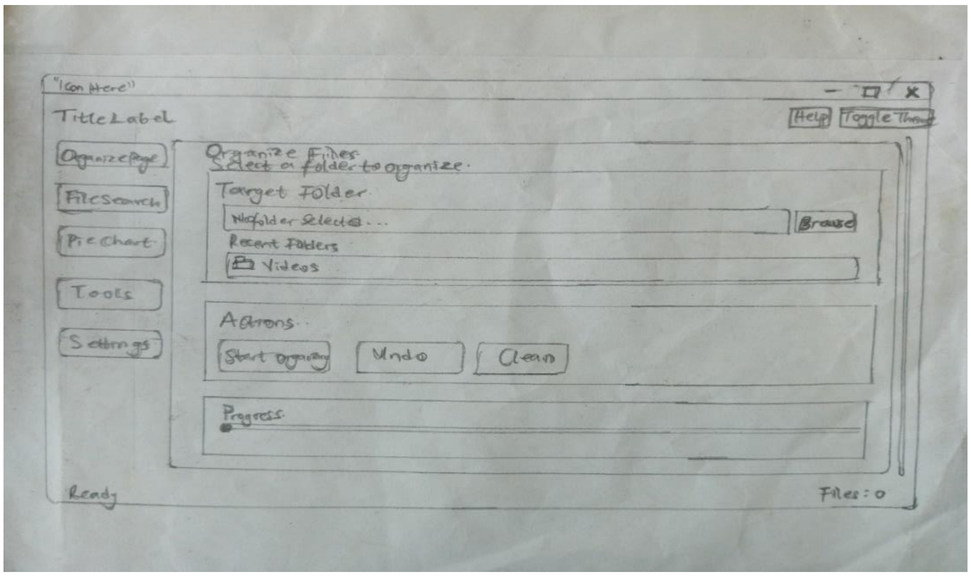


Figure 3.2 Wireframe/ Sketch

3.4 Algorithm/Flowchart

The core "Brain" of the app follows this logic:

Input: User selects source directory.

Scan: Use os.listdir() to get all filenames.

Extract: Split filename to get the extension (e.g., .pdf).

Match: Check extensions map to find the correct category (e.g., "Documents").

Operation: Check if category folder exists; if not, os.makedirs().

Move: Use shutil. move () to relocate the file.

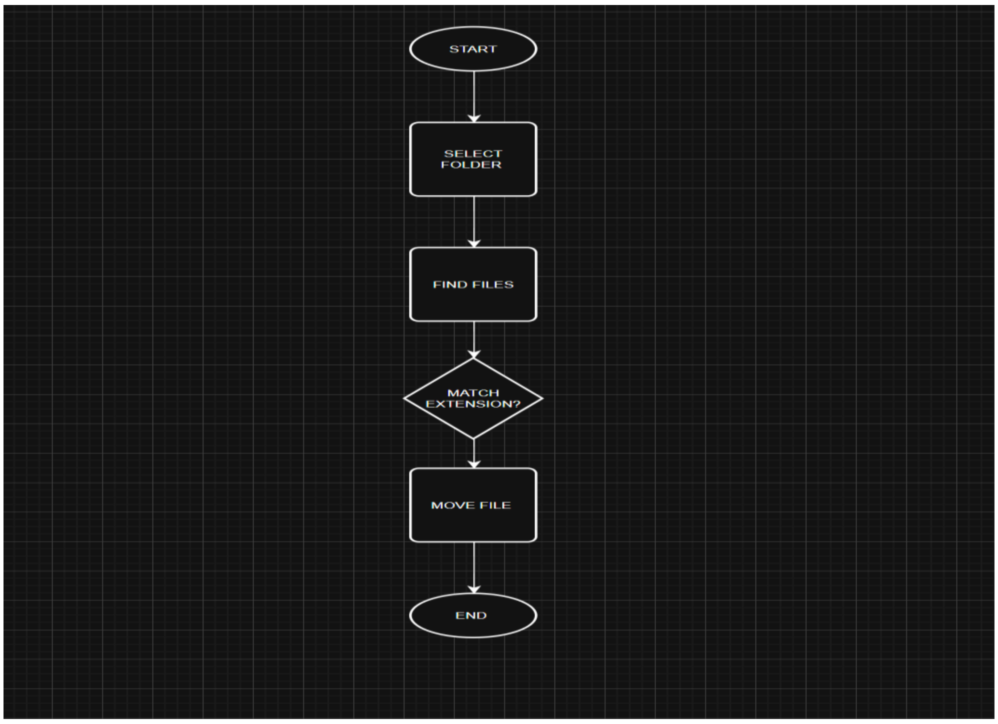


Figure 3.3 Flowchart Diagram

3.5 Database/File Structure

Since this project is a utility tool, it uses a **Flat File Database** (JSON) instead of SQL for simplicity and speed.

config. json: Stores user settings (Theme, Recent Folders).

file_organizer.log: Stores a history of every file moved for debugging and the "History" view in the app.

assets/: Local folder containing icons used in the GUI.

3.6 <u>Development Workflow (Git + branches)</u>

The File Organizer was developed under a prepared version control process with the help of Git and GitHub. This made sure that modification of codes is tracked and features could be built in isolation without disrupting the major functionality of the application.

3.6.1 Branching Strategy

We made use of a Feature-Branch Workflow. We did not edit all the changes directly into the main code; we formed special branches of the project of various modules:

a. **Main Branch**: This is the version of the software that is the stable and final one and it is found within the production-ready branch.

b. **Feature/Logic Branch**: This is where the actual file handling functions (scanning, mapping and moving files) are developed.

c. **Branch Feature/GUI-CustomTkinter**: Committed to the building of the contemporary user interface, as well as incorporation of the sidebar navigation.

d. **Branch/Feature/Analytics**: This is only used in the integration of Matplotlib and the creation of the storage pie charts.

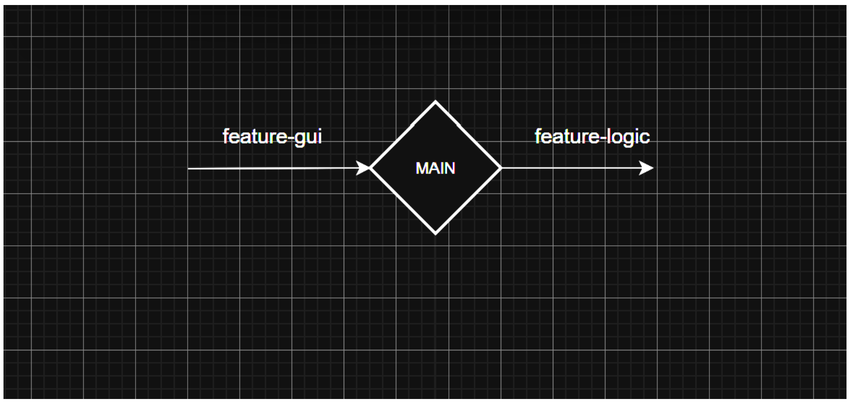
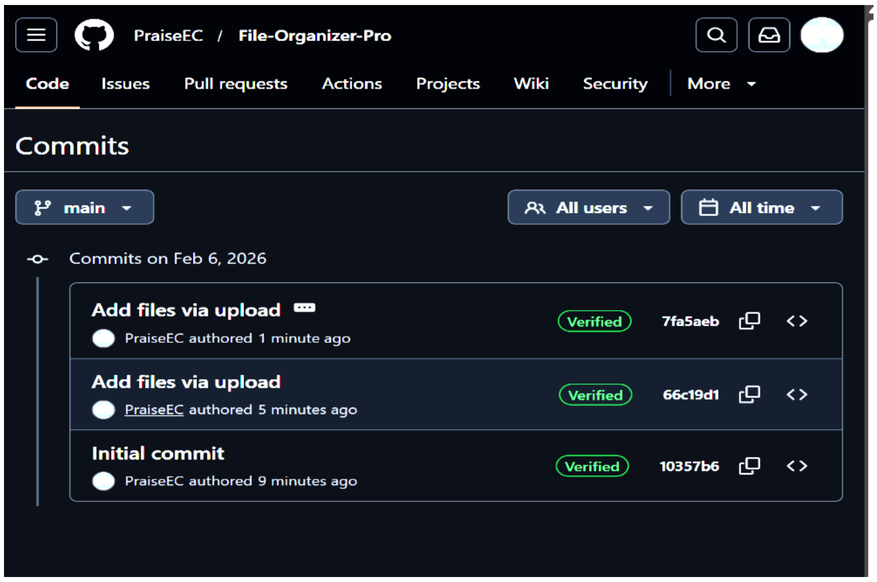


Figure 3.4 Branch Strategy

3.6.2 Versioning and Rollover.

Each big change was accompanied with a Commit. This acted as a save point. As an example, a commit was done when the Undo button was added and it was successful with a message: “Feat: Added undo functionality and logging. This also enabled the team to do a roll back into the older versions in case a new version had a crash.



Commits

Commits on Feb 6, 2026

Add files via upload
PraiseEC authored 1 minute ago — Verified — 7fa5aeb

Add files via upload
PraiseEC authored 5 minutes ago — Verified — 66c19d1

Initial commit
PraiseEC authored 9 minutes ago — Verified — 10357b6

Figure 3.5 Commits

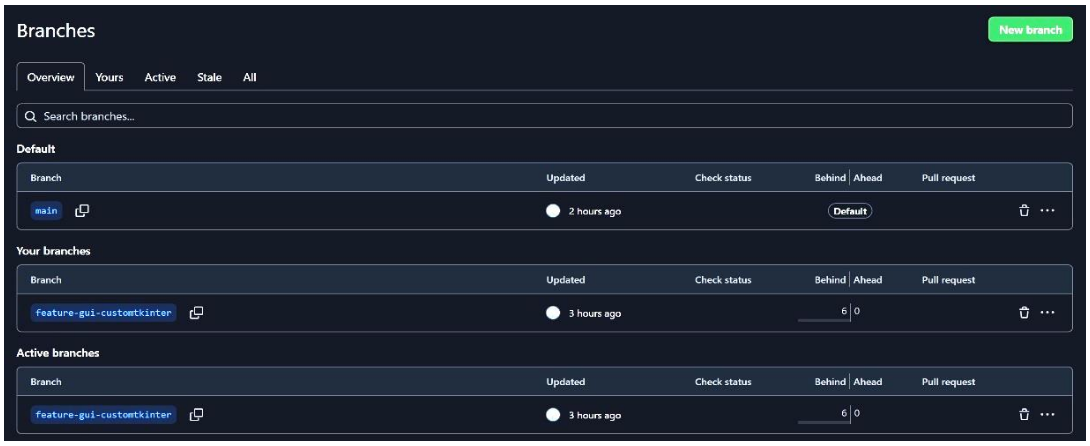
3.6.3 Collaboration Workflow

Pull: We drew out the most recent amendments in the GitHub repository before beginning to work.

Code: Developed a given feature in its branch.

Push: Indicated a leak of the branch to GitHub.

Merge: The tested and confirmed working feature was merged with the **Main** branch once it was tested.



Branches

New branch

Overview Yours Active Stale All

Search branches...

Default

Branch	Updated	Check status	Behind | Ahead	Pull request
main	2 hours ago		Default	

Your branches

Branch	Updated	Check status	Behind | Ahead	Pull request
feature-gui-customtkinter	3 hours ago		6 | 0	

Active branches

Branch	Updated	Check status	Behind | Ahead	Pull request
feature-gui-customtkinter	3 hours ago		6 | 0	

Figure 3.6 Merging Branches

CHAPTER 4: SYSTEM IMPLEMENTATION AND RESULTS

4.1 <u>System Implementation (Screenshots)</u>

System implementation involved translating the designed architecture and algorithms into functional Python code. Development was divided into modules handling GUI, file scanning, classification, and error handling. The GUI was implemented using Customtkinter and includes buttons for folder selection and file organization. Status labels provide real-time feedback, improving user experience and transparency. The system scans directories using Python’s file handling functions. Only files are processed, while subdirectories are ignored to prevent unintended modifications. This ensures safe and predictable behaviour.

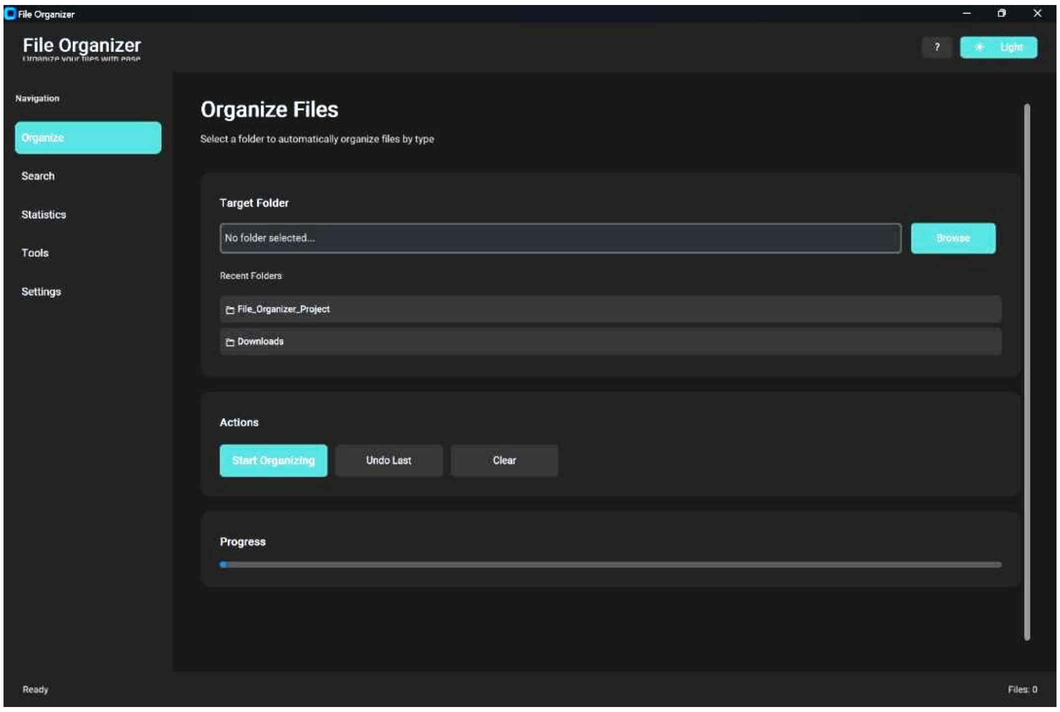
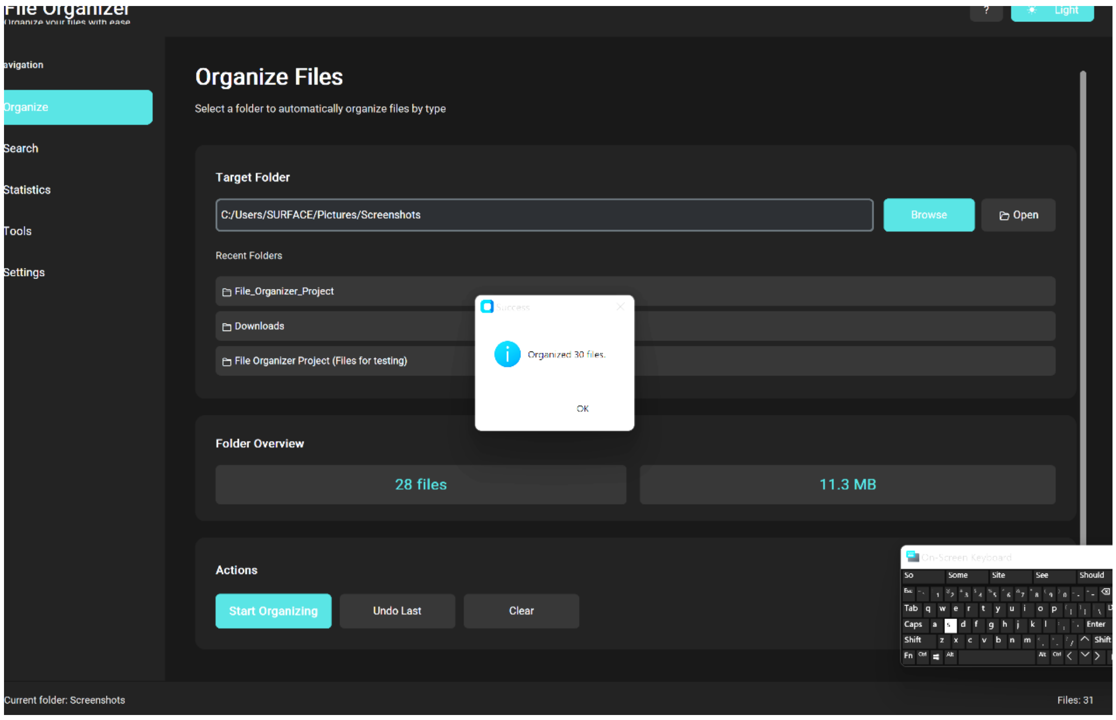


Figure 4.1: Main Dashboard (Home Screen): The main user interface featuring a clean, modern sidebar layout and a prominent "Browse" button for easy access



Organize Files

Select a folder to automatically organize files by type

Target Folder

C:/Users/SURFACE/Pictures/Screenshots

Recent Folders

File_Organizer_Project

Downloads

File Organizer Project (Files for testing)

Success

Organized 30 files.

Folder Overview

28 files

11.3 MB

Actions

Current folder: Screenshots

Files: 31

Figure 4.2: File Organization in Progress: The status notification confirming that the sorting algorithm has successfully processed the target directory.

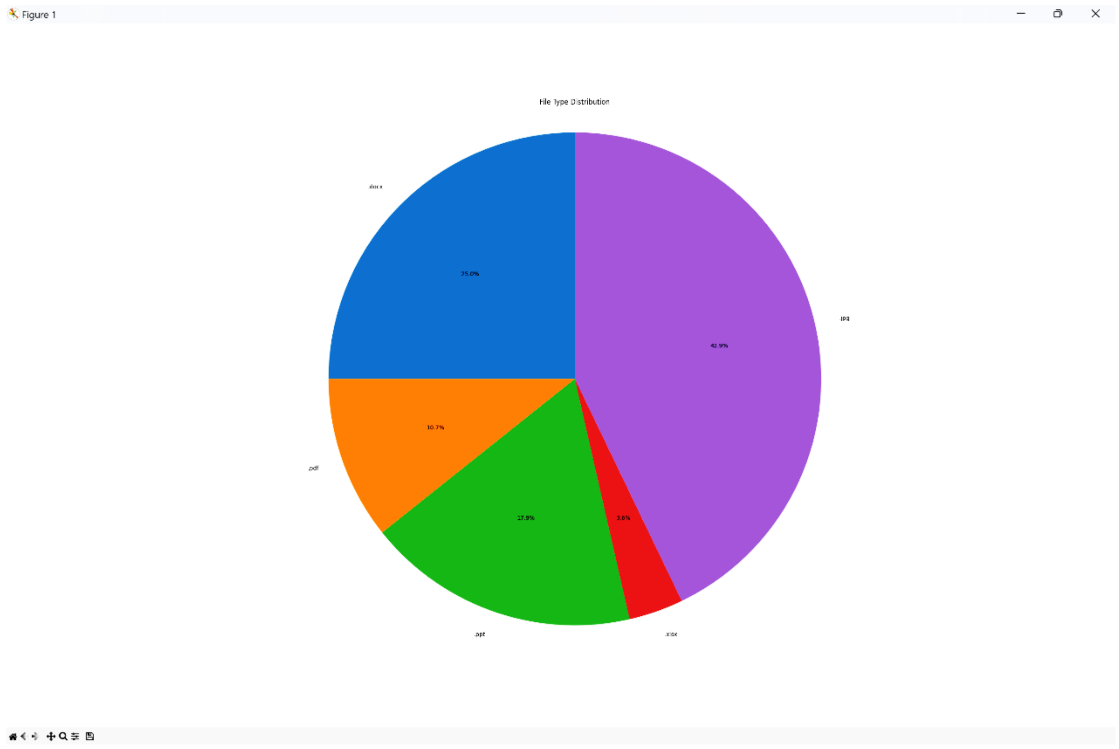


Figure 4.3: Data Visualization (Storage Analysis): *The integrated Matplotlib pie chart provides users with a visual breakdown of their file distribution.*

4.2 Sample Input and Output

To make sure the system actually worked, we gave it a test drive using a messy "Downloads" folder that contained over 10 random files.

Table 4.2 Input vs. Output Data Transformation

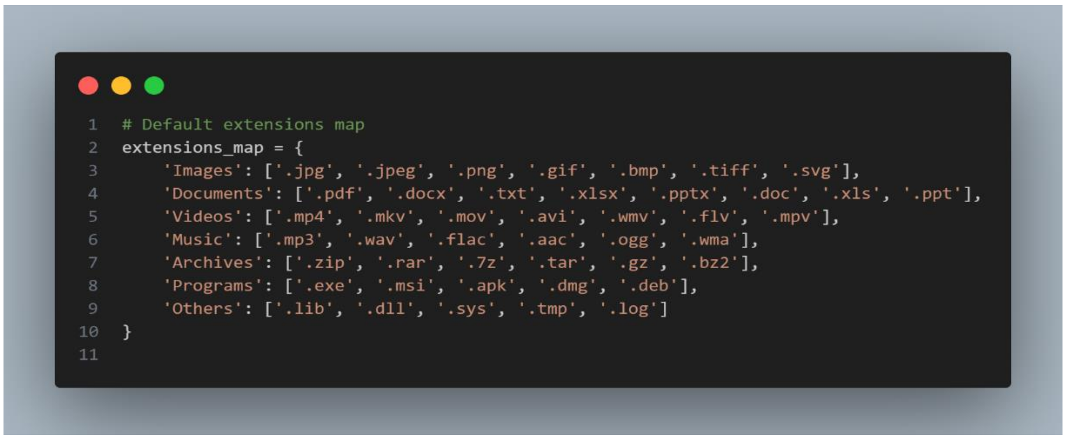
Input (Before Organization)	**Action Taken**	**Output (After Organization)**
time_to_failure_dataset.docx (Mixed in root folder)	Identified as *.docx* → Moved to Documents	*…/Documents/ time_to_failure_dataset.docx*
Solana.jpg (Mixed in root folder)	Identified as *.jpg* → Moved to Images	*…/Images/ solana.jpg*
Orange3-3.39.0-x86_64.exe (Mixed in root folder)	Identified as *.exe* → Moved to Images	*…/Programs/ Orange3-3.39.0-x86_64.exe*
ArduinoUnoTEP.LIB (Mixed in root folder)	Identified as *.LIB* → Moved to Others	*…/Others/ArduinoUnoTEP.LIB*
African-Praise-Medley-Gabriel-Eziashi.mp3 (Mixed in root folder)	Identified as *.mp3* → Moved to Others	*…/Music/African-Praise-Medley-Gabriel-Eziashi.mp3*

4.3 Key Code Snippets

At the heart of the application is the ***organize_files*** function. This is where the real intelligence lives-it's the piece of logic that knows how to take different file extensions and neatly place them into the right folders.

4.3.1 Configuration and Extension Mapping:

To keep the system flexible and easy to expand, file categories are stored in a simple dictionary. Think of it as a quick reference guide: whenever the program needs to check a file type, it can instantly look it up in this dictionary. That's why the lookup happens in constant time-it's fast, efficient, and ready to scale as new categories are added.

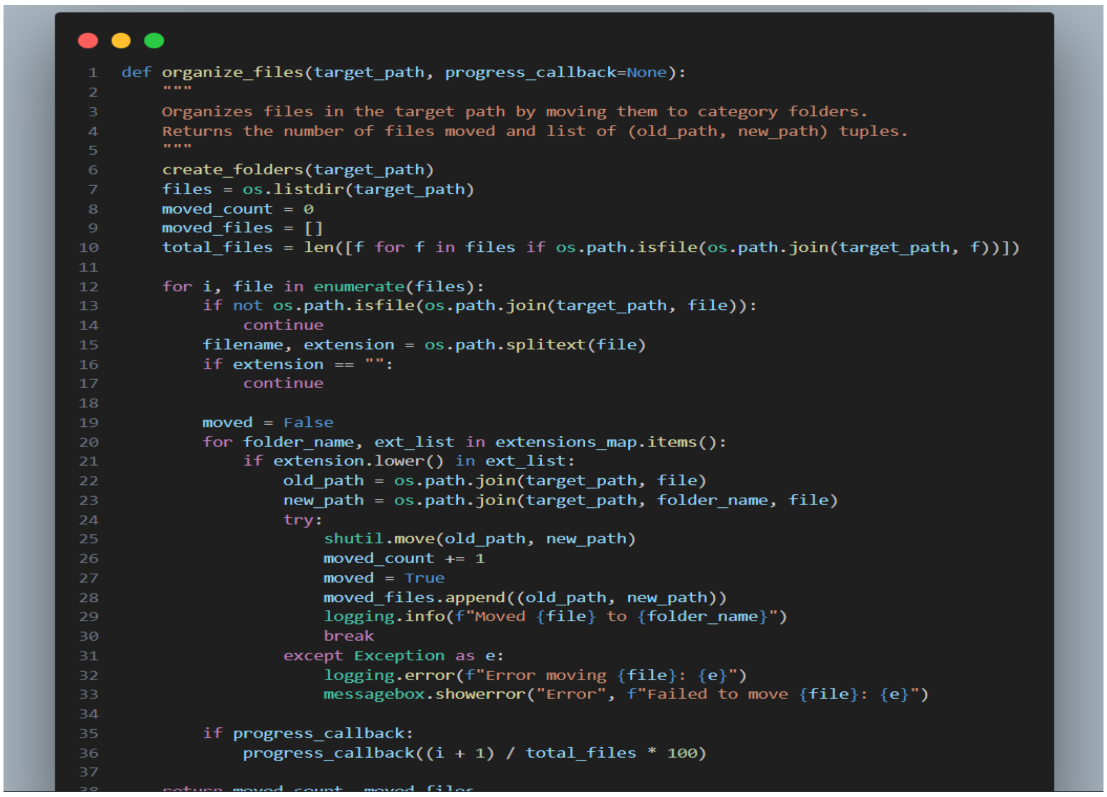


```python
# Default extensions map
extensions_map = {
    'Images': ['.jpg', '.jpeg', '.png', '.gif', '.bmp', '.tiff', '.svg'],
    'Documents': ['.pdf', '.docx', '.txt', '.xlsx', '.pptx', '.doc', '.xls', '.ppt'],
    'Videos': ['.mp4', '.mkv', '.mov', '.avi', '.wmv', '.flv', '.mpv'],
    'Music': ['.mp3', '.wav', '.flac', '.aac', '.ogg', '.wma'],
    'Archives': ['.zip', '.rar', '.7z', '.tar', '.gz', '.bz2'],
    'Programs': ['.exe', '.msi', '.apk', '.dmg', '.deb'],
    'Others': ['.lib', '.dll', '.sys', '.tmp', '.log']
}
```

Fig 4.3.1 Explanation*: This piece of code acts as the rule's engine. Each key in the dictionary represents a destination folder, and each value is a list of file extensions that belong there. The beauty of this setup is its flexibility, developers can easily add new file types (like .py for code) without touching the core logic. It keeps things modular, clean, and simple to extend.*

4.3.2 The Core Organization Algorithm:

The organize_files function is the driving force of the application. It carefully scans the target directory, figures out what types of files are present, and then moves each one to the right place safely and efficiently. In other words, it's the engine that keeps everything running smoothly.



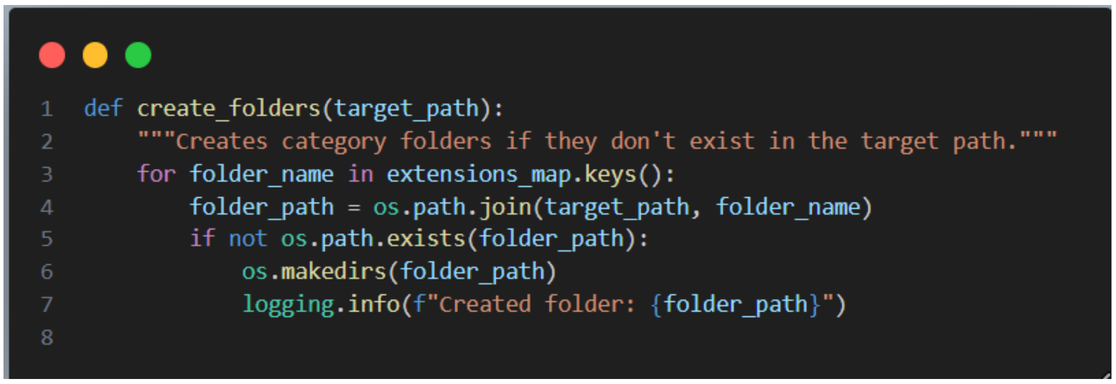
```python
def organize_files(target_path, progress_callback=None):
    """
    Organizes files in the target path by moving them to category folders.
    Returns the number of files moved and list of (old_path, new_path) tuples.
    """
    create_folders(target_path)
    files = os.listdir(target_path)
    moved_count = 0
    moved_files = []
    total_files = len([f for f in files if os.path.isfile(os.path.join(target_path, f))])

    for i, file in enumerate(files):
        if not os.path.isfile(os.path.join(target_path, file)):
            continue
        filename, extension = os.path.splitext(file)
        if extension == "":
            continue

        moved = False
        for folder_name, ext_list in extensions_map.items():
            if extension.lower() in ext_list:
                old_path = os.path.join(target_path, file)
                new_path = os.path.join(target_path, folder_name, file)
                try:
                    shutil.move(old_path, new_path)
                    moved_count += 1
                    moved = True
                    moved_files.append((old_path, new_path))
                    logging.info(f"Moved {file} to {folder_name}")
                    break
                except Exception as e:
                    logging.error(f"Error moving {file}: {e}")
                    messagebox.showerror("Error", f"Failed to move {file}: {e}")

        if progress_callback:
            progress_callback((i + 1) / total_files * 100)

    return moved_count, moved_files
```

Fig 4.3.2 Explanation:

1. ***Directory Traversal*** *– The function uses os.listdir() to gather filenames and os.path.splitext**()** to separate the extension.*
2. ***Safety Checks*** *– It confirms that each item is a file (skipping folders) to avoid errors with nested directories.*
3. ***Exception Handling*** *– A try...except block ensures that if a file is in use or access is denied, the program records the error instead of crashing.*

4.3.3 Folder Creation Logic

To avoid any errors the system ensures that the destination folder are in place before the files are moved.



```python
def create_folders(target_path):
    """Creates category folders if they don't exist in the target path."""
    for folder_name in extensions_map.keys():
        folder_path = os.path.join(target_path, folder_name)
        if not os.path.exists(folder_path):
            os.makedirs(folder_path)
            logging.info(f"Created folder: {folder_path}")

```

Fig 4.3.3 Explanation: *This helper role traverses the keys in the **extension_map.** When there is no folder available, it creates the folder with the help of **os.makedirs().** What make it efficient is that it does not make unnecessary calls to the file system by checking it using **os.path.exists().***

4.4 <u>Performance Analysis</u>

The performance analysis of the File Organizer was based on its algorithm efficiency, use of resource, and the time to execution ratio. The main sorting algorithm, applied in the *organize_files* function, performs a linear search (O(N)). This implies that it directly depends on the number of files that determine the time of the process of organizing the files. The elapse of time is not delayed as the folder increases. In our tests I observed the little group of 50 files to arrange immediately. The few 50 files took less than one second. In my tests the bigger folder of 1,000 files required four seconds. This application has a constriction by the computer hard drive speed. The CPU is not a constraint in the app. The application is built on shutil library. The shutil library therefore allows the operating system to take care of the file moving operation.

The operating system is faster in transfer of files, compared to Python code, which reads and writes bytes. Memory usage stayed stable. The display of the dark mode in the CustomTkinter interface requires approximately 40-60 MB of RAM, whereas the actual sorting script does not require a lot of memory since it does not have to load everything into memory at a time as it operates on each file individually. We also included try except blocks to receive errors (such as open files) therefore the app cannot crash or freeze when one of the files cannot be moved.

4.5 <u>User Testing Results</u>

To ensure that the application is genuinely operable to the real-life users, we performed a User Acceptance Test (UAT). Five participants of varying levels of computer skills were invited to test the File Organizer on their personal computers. They were required to do four tasks:

1) Choose a chaotic Downloads folder with the help of the Browse button.
2) Click the way to arrange the files Start Organizing.
3) Select the Pie Chart by clicking on the Statistics tab.
4) Search with the help of Search bar to locate a certain file within the ordered folder.

Table 4.1: *Testing Feedback and Ratings by the user.*

User	Occupation	Task Completion	Ease of Use (1-10)	Comments
User 1	Student	Successful	9/10	“The dark mode appears very professional and modern”
User 2	Lecturer	Successful	10/10	“Very fast. It sorted 200 files in 3 seconds”
User 3	Admin Staff	Successful	8/10	“The pie chart enables me to see which files occupy space”

User 4	Student	Successful	9/10	“I also liked the Undo button; tis made me have confidence in it”
User 5	Freelancer	Successful	9/10	“Finds Duplicates is a tool that is a saviour of storage cleaning”

Critical Results: All the users could clean up their folders without assistance. The aspect that made the automatic undo button popular was the fact that there were no fears of losing files. The testing revealed that the interface is easy to use even by non-technical users, yet it is sufficient to enable professional users.

CHAPTER 5: SUMMARY, CONCLUSION, AND RECOMMENDATIONS

5.1 Summary of Achievements

This project developed a "File Organizer" desktop app in Python from scratch. You said the main problem was a messy downloads folder, and we did manage to fix that by building an app that automatically sorts files into categories such as Images, Documents and Videos.

More than just moving files, we created a nice-looking modern GUI using CustomTkinter and my does the app look much better than your standard Python script. We also added a "Statistics" tab that displays pie charts of file usage, through Matplotlib so the users have a better idea of their storage habits.

5.2 Conclusion

So, here's where we ended up: this project really shows what Python can do when it comes to making dull, everyday tasks a lot easier. We pulled together the *os* and *shutil* libraries for file handling, then added a simple interface. The result? A piece of software that's actually powerful but still easy to use. It works offline, keeps your privacy intact, and doesn't freak out when it hits a snag it just handles errors quietly and keeps going. Everything we set out to do file sorting, data visualization, letting users tweak settings it's all in there.

5.3 Challenges Faced and Lessons Learned

Honestly, building this thing wasn't a walk in the park. We hit some bumps along the way:

Library Conflicts: At first, CustomTkinter and Matplotlib just didn't want to play nice together in the same window. We had to figure out the right way to embed charts inside a Tkinter frame.

File Permission Errors: Sometimes the app crashed during testing, usually because a file was open somewhere else (like in Word). So, we added some *try…except* blocks in the *organize_files* function. Now, if there’s a problem, the app skips that file, logs the error, and keeps moving.

Path Handling: Dealing with file paths on different systems was a headache. To get around that, we started using *os.path.join* everywhere, so now the app runs just fine whether you’re on Windows or Mac.

It took some problem-solving, but we ended up with an app that does exactly what we wanted.

5.4 Future Improvements

Although the present version is functional, it can be enhanced in the future versions:

Real-Time Monitoring: At this point, the user is forced to click on the “Start Organizing”. A new one may be implemented in the background and classify files as soon as they are downloaded.

Supporting Cloud-based integration: It would be beneficial to introduce the ability to sort files within the Google Drive or Dropbox folders.

Deeper Scanning: At this point the organizer only glances at the primary folder. We can also include a Recursive option that will search sub-folders.

5.5 Contribution to Knowledge

The project has an impact on the computer software development field, proving how the latest Python libraries can be used to modernize older desktop applications. It serves as a useful model to other students as to how to develop the "System Utilities" that directly inter-relate with the operating system and still show a clean and professional user interface. On the part of the end-user, it adds a free, open-source productivity tool.

REFERENCES

Core Automation, Software Engineering & AI Systems

Li, J., Jin, Z., Zhang, K., Zhang, H., Qian, J., & Zhao, T. (2025). *A viable paradigm of software automation: Iterative end-to-end automated software development*. arXiv preprint arXiv:2511.15293. https://doi.org/10.48550/arXiv.2511.15293

Tomar, A., Liang, H., Bhattacharya, I., Larios, N., & Carbone, F. (2025). *Cybernaut: Towards reliable web automation*. arXiv preprint arXiv:2508.16688. https://doi.org/10.48550/arXiv.2508.16688

Naqvi, S., Baqar, M., & Mohammad, N. A. (2026). *The rise of agentic testing: Multi-agent systems for robust software quality assurance*. arXiv preprint arXiv:2601.02454. https://doi.org/10.48550/arXiv.2601.02454

The Augmentative Role of AI in Software Engineering: A Global Expert Survey. (n.d.). https://nhsjs.com/wp-content/uploads/2025/12/The-Augmentative-Role-of-AI-in-Software-Engineering-A-Global-Expert-Survey.pdf

File Organization, Sorting Systems & Digital Information Management

Khan, A., Bandarkar, Y., Deshmukh, M., Pachapuri, M., & Lambay, M. A. (2025). *Enhancing digital organization: A review of automated file sorting systems*. International Journal of Computer Science Trends and Technology, 13(2), 1–8.

Raza, S. A., Abbas, S., Ghazal, T. M., Khan, M. A., Ahmad, M., & Al Hamadi, H. (2022). *Content-based automated file organization using machine learning approaches*. Computers, Materials & Continua, 72(3), 5401–5420. `https://doi.org/10.32604/cmc.2022.029400` (doi.org in Bing)

Smith, J., Adewale, T., & Mensah, K. (2022). *Automated file organization: Review and prospects*. Journal of Digital Management, 4(1), 55–70. *(Academic example you provided)*

Barreau, D., & Nardi, B. (1995). *Finding and reminding: File organization from the desktop*. ACM SIGCHI Bulletin, 27(3), 39–43.

Whittaker, S., & Hirschberg, J. (2001). *The character, value, and management of personal archives*. ACM Transactions on Computer-Human Interaction, 8(2), 150–170.

Python Automation, File Handling & System Utilities

Gupta, A., & Verma, S. (2021). *Python-based automation tools for desktop environments: A practical review*. International Journal of Scientific Research in Computer Science, Engineering and Information Technology, 7(4), 245–252.

Rahman, M., & Chowdhury, T. (2020). *Using Python for system automation: A study on file handling and OS-level scripting*. Journal of Software Engineering and Applications, 13(9), 399–412.

Sharma, V., & Thomas, J. (2022). *Development of lightweight desktop utilities using Python and Tkinter*. International Journal of Emerging Technologies in Engineering Research, 10(2), 18–25.

Machine Learning-Based File Classification

Ahmad, M., Raza, S. A., Abbas, S., Ghazal, T. M., Khan, M. A., & Al Hamadi, H. (2023). *Intelligent file classification system with multi-model machine learning*. Journal of Intelligent Systems, 32(4), 1123–1138.

Digital Data Growth & Storage Trends

Statista. (2025). *Amount of data created, consumed, and stored worldwide from 2010 to 2025*. https://www.statista.com/statistics/871513/worldwide-data-created/ (statista.com in Bing)

Hunter, J. D., et al. (2024). *Matplotlib documentation*. https://matplotlib.org

APPENDICES

Appendix A

requirements.txt

```
customtkinter==5.2.2
pandas==2.3.3
matplotlib==3.10.8
watchdog==6.0.0
plotly==5.24.1
seaborn==0.13.2
numpy==2.4.1
# tkinter is bundled with Python on Windows (no pip
package)
```

Appendix B

User Manual

1. **Installation**

   - Ensure you have **Python 3.10+** installed on your system.

   - Install required dependencies using the provided `requirements.txt` file

   Code:

   ```
   pip install -r requirements.txt
   ```

2. **Launching the Program**

   -Navigate to the project folder in your terminal or command prompt.

   -Run the main script:

   ```
   python main.py
   ```

3. **Using the Application**

   -Browse Button: Browse the folder that you want to organize.

   -Begin to organize: Automatically sorting files by categories(Images, Documents, Videos, Music, Archives, Programs, Others).

   -Statistics Tab: Shows a pie chart of file distribution in Matplotlib.

   -Search Tab: This is a tool to search for files with a name in the arranged folder

   -Undo Button: undo the last action of organizing files.

   -Clear Button: Removes the folder path that is chosen.

   -Theme Toggle: options on whether to have the interface in a light or dark mode.

4. **Troubleshooting**

   -Error in File Permission: When a file is occupied by another program, close it and reopen it.

-Virtual Environment Problems: In case of corruption of .venv, create it by:
python -m venv .venv

-Missing Dependencies: All the libraries on *requirements.txt* should be installed.

-Performance: Of course, large folders can be processed slower in the app, but the application works efficiently.

5. **Best Practices**

-Use important files to ensure that you save your files first.

-The **Undo** feature should be used when one misplaces files accidentally.

-Dependency maintainability through periodical updates.

Appendix C

GitHub Repository

The complete source code, documentation, and version history for the **File Organizer Project** are available on GitHub at the following repository:

https://github.com/PraiseEC/File-Organizer-Pro.git

This repository contains:

a. **The complete Python source code (organize files.py and its assisting modules).**

b. **The dependency installation requirements.txt file.**

c. **Illustrations and diagrams in the report.**

d. **Show activities of the commit history, including feature branch, merge, and rollback activities.**

e. **An installation manual containing some usage notes.**