PharmaSUG 2026 - Paper TT-175

Enhancing Quality and Efficiency in Clinical Programming with a Python-Based Automated File Comparison Tool

Ratheesh Gunda, Kite Pharma

ABSTRACT

Ensuring consistency and accuracy of files across directories and sub-directories is a critical step in statistical programming workflows to ensure that the latest datasets are used for analysis. Discrepancies in file presence or versioning can lead to outdated analyses and quality issues. This paper introduces a Python-based automated file comparison tool with a graphical user interface (GUI) designed to streamline the validation process between hierarchical folder structures and flat directories.

The tool performs three core functions:
1. Verifies file presence or absence between folders,
2. Detects timestamp changes to identify updated or modified files, and
3. Generates a detailed Excel report summarizing differences for easy review and documentation.

With its intuitive GUI, users can select source and target folders, initiate comparisons, and instantly review results, eliminating the need for manual checks or repetitive library setups. Built using Python's standard libraries and common open-source packages, the tool efficiently traverses subdirectories, extracts file metadata, and presents results in a clear, audit-ready format that highlights missing, modified, or newly added files. Unlike traditional programming workflows that require manual library management for multiple subfolders, this Python solution offers greater flexibility and automation, allowing large-scale comparisons to be completed within minutes. By reducing manual effort and enhancing traceability, the tool supports regulatory compliance, data integrity, and quality control across programming teams.

This paper will describe the tool's design, workflow, and reporting capabilities, providing practical guidance for implementing automated file validation processes to improve efficiency and reproducibility in programming.

INTRODUCTION

Clinical programming workflows require accurate and consistent datasets. Manual validation across directories is inefficient and error-prone, especially in large studies. Programming workflows often involve multiple directory structures, including raw data, intermediate datasets, and analysis-ready files. This paper presents an automated Python-based solution to improve efficiency and reliability.

BACKGROUND AND PROBLEM STATEMENT

During analysis workflows, programmers are often required to validate that datasets in the analysis folder are consistent with source data received from multiple external vendors. In many cases, vendor data is stored across separate subdirectories, while the datasets for analysis are maintained in a single, centralized location.

A common approach to validation involves manually opening the analysis folder alongside each vendor subfolder and performing side-by-side comparisons of file names and timestamps. This process is tedious and difficult to manage, particularly when dealing with multiple vendors and a

large number of files. The likelihood of overlooking missing or updated files increases as complexity grows.

An alternative method involves developing programs to perform file comparisons. While more structured, this approach is labor-intensive to implement and maintain. Programmers must write or update programs based on folder structure and comparison logic and create additional steps to generate summary outputs, often in Excel format. In addition, these programs require ongoing updates to accommodate new files or changes in folder structures, further increasing the overall effort.

Both approaches rely heavily on manual intervention and are not easily scalable and increase the risk of inconsistencies between source data and datasets in the analysis folder.

Figure 1 illustrates the folder structures of a hierarchical structure (one-level down) where equivalent files are distributed across multiple subdirectories.

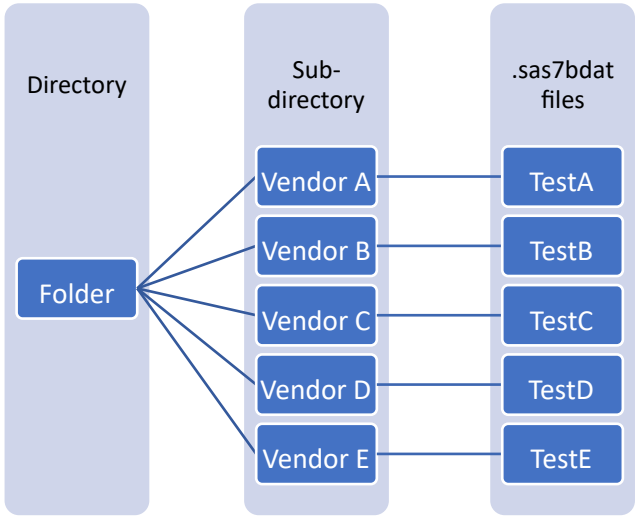

Figure1

Figure 2 below illustrates the flat folder structure with multiple files within it.

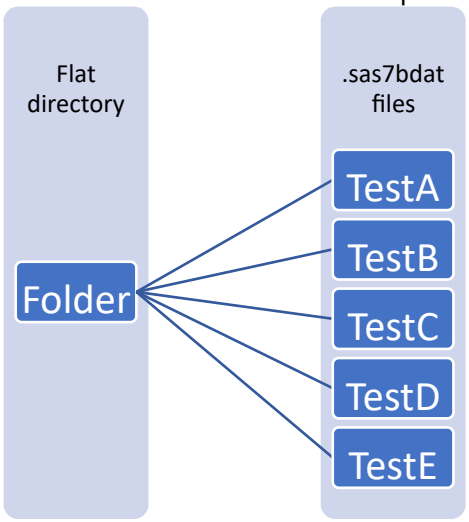

Figure 2

Another scenario arises when both the source (shown in blue) and analysis directories (shown in green) contain files organized in flat structures (without subfolders), with potential differences in file types across the two locations as shown below. These limitations highlight the need for an automated, scalable, and user-friendly solution for file validation.

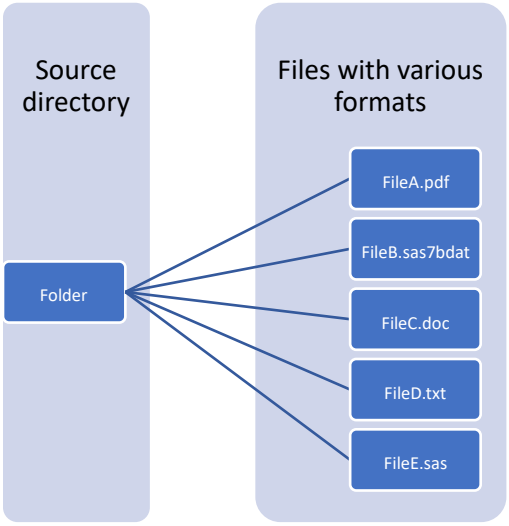

Figure 3

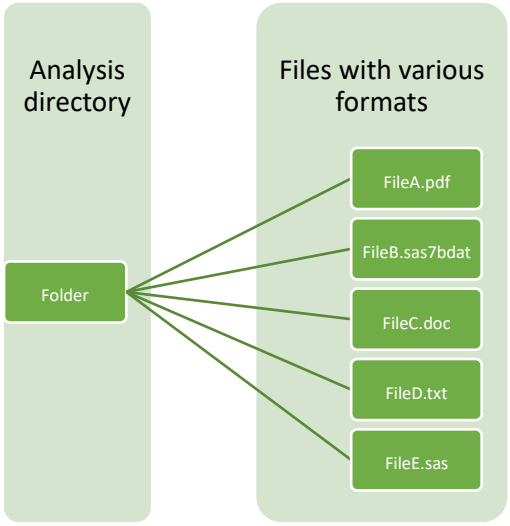

Figure 4

OVERVIEW OF TOOL

The tool enables automated file comparison and report generation with minimal user interaction, requiring only a few clicks to select input folders and execute the comparison. To address different directory comparison scenarios, the application consists of two primary tabs:

- **Tab 1:**

  Compares files with same format between a flat directory and files located multiple subdirectories.

- **Tab 2:**

  Compares files between two directories, supporting both similar and different file formats.

Additional key features of the tool include:

- User-friendly graphical interface developed using Python's *tkinter* library
- File validation based on timestamp and file size
- Support for extension-based filtering and handling of hidden files
- Automated generation of Excel reports with summary outputs
- Color-coded results to facilitate quick identification of differences

HOW TO USE THIS TOOL

The tool provides an intuitive graphical interface with 2 functional tabs. After installing Python, follow the steps below:

I. Launching and Selecting Tabs:
   a. Users start by launching the tool via double-clicking the executable.
   b. Select the appropriate tab based on the comparison scenario required.

II. Folder Selection Process:
   a. Choose both folders to compare by clicking the 'Select Folder' buttons.
   b. Confirm the displayed folder paths.

III. Comparison Execution:
   a. Click the 'Compare' button to automatically scan directories and perform comparisons.
   b. Result Display and Interpretation:
      i. Comparison results appear in a list box with color-coded statuses:
         1. Green → No changes
         2. Orange → Modified files
         3. Red → Missing files IV.

Exporting Report:

a. Users can select a location to save the comparison excel report for documentation.

Use Cases

The tool is suitable for multiple file validation scenarios, including:

- Ensuring all datasets are properly transferred:
  - Pre-analysis checks ○ Development vs. production folders
- Comparison across different stages of analysis:
  - Define vs. Prep vs. M5 folders

Limitations

While effective, the tool has some limitations:

- Comparison is based on metadata (timestamp, size) rather than actual file content
- Internal data changes cannot be detected if timestamps remain unchanged.

Conclusion

The Python-based automated file comparison tool provides a practical and efficient solution for validating files across directory structures in clinical programming workflows. By reducing manual effort and program maintenance, improving accuracy, and enhancing traceability, the tool supports data integrity and regulatory compliance.

Furthermore, the tool also enables standardized reporting, ensuring consistent and audit-ready outputs for review and documentation. Its flexible design and user-friendly interface make it suitable for a wide range of use cases, allowing programming teams to streamline file validation processes and focus on higher-value tasks.

References

- Python Software Foundation. Python Language Reference, version 3.13. Available at http://www.python.org

- Generative AI tools, including ChatGPT-5 mini (OpenAI), copilot were used to assist with code, and refinement of the paper.

Acknowledgements

- I gratefully acknowledge the review, assistance of Jessie Teng, Kai Chen, and support of my family.

Contact information

- Name: Ratheesh Gunda
- Email: Rgunda114@gmail.com

Appendix

```
import os
import tkinter as tk
from tkinter import filedialog, ttk, messagebox
from datetime import datetime
import pandas as pd
from openpyxl import load_workbook
from openpyxl.styles import Font

# ---------- File listing one level below folder ----------
def list_sas_files_one_level(folder):
    """
    Lists .sas7bdat files one level below the folder (in immediate subfolders only).
    Returns dict: filename -> datetime(mtime)
    """
    sas_files = {}
    if not folder or not os.path.isdir(folder):
        return sas_files
    try:
        with os.scandir(folder) as entries:
            for entry in entries:
                if entry.is_dir():
                    try:
                        with os.scandir(entry.path) as subentries:
                            for sub in subentries:
                                if sub.is_file() and sub.name.lower().endswith(".sas7bdat"):
                                    try:
                                        stat = sub.stat()
                                        sas_files[sub.name] = datetime.fromtimestamp(stat.st_mtime)
                                    except Exception as e:
                                        print(f"Error reading {sub.path}: {e}")
                    except Exception as e:
                        print(f"Error reading subfolder {entry.path}: {e}")
    except Exception as e:
        print(f"Error reading folder (one-level) {folder}: {e}")
    return sas_files

# ---------- Global file listing ----------
def parse_extensions_filter(filter_text):
    if not filter_text or filter_text.strip() in ["*.*", "*"]:
        return None
    parts = [p.strip().lower() for p in filter_text.split(",") if p.strip()]
    exts = set()
    for p in parts:
        if p.startswith("*."):
            exts.add(p[1:])
        elif p.startswith("."):
            exts.add(p)
        else:
            exts.add("." + p)
    return exts if exts else None

def list_all_files(folder, recursive=True, extensions=None, ignore_hidden=True):
    files = {}
    if not folder or not os.path.isdir(folder):
        return files
```

```
    def is_hidden(name: str) -> bool:
        return name.startswith(".")

    try:
        if recursive:
            for root, dirs, filenames in os.walk(folder):
                if ignore_hidden:
                    dirs[:] = [d for d in dirs if not is_hidden(d)]
                for fname in filenames:
                    if ignore_hidden and is_hidden(fname):
                        continue
                    full_path = os.path.join(root, fname)
                    try:
                        stat = os.stat(full_path)
                    except Exception:
                        continue
                    ext = os.path.splitext(fname)[1].lower()
                    if extensions is not None and ext not in extensions:
                        continue
                    rel_path = os.path.relpath(full_path, folder)
                    files[rel_path] = {
                        'size': stat.st_size,
                        'mtime': datetime.fromtimestamp(stat.st_mtime),
                        'abs_path': full_path
                    }
        else:
            with os.scandir(folder) as entries:
                for entry in entries:
                    if not entry.is_file():
                        continue
                    if ignore_hidden and is_hidden(entry.name):
                        continue
                    ext = os.path.splitext(entry.name)[1].lower()
                    if extensions is not None and ext not in extensions:
                        continue
                    try:
                        stat = entry.stat()
                        files[entry.name] = {
                            'size': stat.st_size,
                            'mtime': datetime.fromtimestamp(stat.st_mtime),
                            'abs_path': entry.path
                        }
                    except Exception as e:
                        print(f"Error reading file {entry.path}: {e}")
    except Exception as e:
        print(f"Error walking folder {folder}: {e}")

    return files

# ---------- Tab 1 comparison ----------
def compare_sub_vs_flat_display(import_dict, raw_other_dict, threshold_seconds=1):
    results = []
    for key in sorted(import_dict.keys()):
        base_time = import_dict[key]
        if key in raw_other_dict:
            compare_time = raw_other_dict[key]
            diff = abs((base_time - compare_time).total_seconds())
            changed = "Yes" if diff > threshold_seconds else "No"
            exists = "Yes"
            color = "orange" if changed == "Yes" else "green"
        else:
            exists = "No"
            changed = "NA"
            color = "red"
        results.append((key, exists, changed, color))
    return results

# ---------- Tab 2 comparison ----------
def compare_all_files_top_level(dictA, dictB, threshold_seconds=1):
    rows = []
    keys = set(dictA.keys()) | set(dictB.keys())
```

```
    for fname in sorted(keys):
        A = dictA.get(fname)
        B = dictB.get(fname)
        if A and B:
            size_same = (A['size'] == B['size'])
            time_diff = abs((A['mtime'] - B['mtime']).total_seconds())
            time_same = (time_diff <= threshold_seconds)
            if size_same and time_same:
                status = "Same"
                color = "green"
            else:
                status = "Changed"
                color = "orange"
            rows.append((
                fname,
                status,
                A['size'],
                B['size'],
                A['mtime'].strftime("%Y-%m-%d %H:%M"),
                B['mtime'].strftime("%Y-%m-%d %H:%M"),
                color
            ))
        elif A and not B:
            rows.append((fname, "Only in A", A['size'], "", A['mtime'].strftime("%Y-%m-%d %H:%M"), "", "red"))
        elif B and not A:
            rows.append((fname, "Only in B", "", B['size'], "", B['mtime'].strftime("%Y-%m-%d %H:%M"), "red"))
    return rows

# ---------- GUI ----------
class FolderComparerApp:
    def __init__(self, root):
        self.root = root
        self.root.title("File Comparison Tool (SAS & Any Files)")

        style = ttk.Style()
        style.configure("Custom.TNotebook.Tab", font=('Helvetica', 12, 'bold'), foreground='blue')

        self.tabControl = ttk.Notebook(root, style="Custom.TNotebook")
        self.tab1 = ttk.Frame(self.tabControl)
        self.tab3 = ttk.Frame(self.tabControl)

        self.tabControl.add(self.tab1, text='1. Compare Sub folder vs Flat folder')
        self.tabControl.add(self.tab3, text='2. Compare All Files')
        self.tabControl.pack(expand=1, fill="both")

        self.create_tab1()
        self.create_tab2()

    # ----- Tab 1 -----
    def create_tab1(self):
        frame = ttk.Frame(self.tab1, padding=10)
        frame.pack(fill='both', expand=True)

        self.import_path = tk.StringVar()
        self.raw_other_path = tk.StringVar()

        ttk.Button(frame, text="Select Sub Folder",
                   command=lambda: self.select_folder(self.import_path)).pack()
        ttk.Label(frame, textvariable=self.import_path).pack()

        ttk.Button(frame, text="Select Flat Folder",
                   command=lambda: self.select_folder(self.raw_other_path)).pack()
        ttk.Label(frame, textvariable=self.raw_other_path).pack()

        ttk.Button(frame, text="Compare", command=self.compare_import_vs_raw).pack(pady=5)
        ttk.Button(frame, text="Save Report to Excel", command=self.save_results_to_excel).pack(pady=5)

        columns_tab1 = ("Filename", "Exists in Flat folder", "Diff. in Timestamp")
        self.result_tree1 = ttk.Treeview(frame, columns=columns_tab1, show="headings", height=20)
        for col in columns_tab1:
            self.result_tree1.heading(col, text=col)
            self.result_tree1.column(col, width=320 if col == "Filename" else 220)
```

```
        self.result_tree1.pack(fill='both', expand=True)

    # ----- Tab 2 -----
    def create_tab2(self):
        frame = ttk.Frame(self.tab3, padding=10)
        frame.pack(fill='both', expand=True)

        self.anyA_path = tk.StringVar()
        self.anyB_path = tk.StringVar()
        self.any_ignore_hidden = tk.BooleanVar(value=True)
        self.any_ext_filter = tk.StringVar(value="*.*")

        path_row = ttk.Frame(frame)
        path_row.pack(fill='x', pady=5)
        ttk.Button(path_row, text="Select Folder A",
                   command=lambda: self.select_folder(self.anyA_path)).pack(side='left')
        ttk.Label(path_row, textvariable=self.anyA_path).pack(side='left', padx=10)

        path_row2 = ttk.Frame(frame)
        path_row2.pack(fill='x', pady=5)
        ttk.Button(path_row2, text="Select Folder B",
                   command=lambda: self.select_folder(self.anyB_path)).pack(side='left')
        ttk.Label(path_row2, textvariable=self.anyB_path).pack(side='left', padx=10)

        opts_row = ttk.Frame(frame)
        opts_row.pack(fill='x', pady=5)
        ttk.Checkbutton(opts_row, text="Ignore hidden (.)", variable=self.any_ignore_hidden).pack(side='left')
        ttk.Label(opts_row, text="Extension filter:").pack(side='left', padx=(20, 5))
        ttk.Entry(opts_row, textvariable=self.any_ext_filter, width=30).pack(side='left')

        ttk.Button(frame, text="Compare", command=self.compare_any_files_top_level).pack(pady=5)
        ttk.Button(frame, text="Save Report to Excel", command=self.save_results_to_excel).pack(pady=5)

        columns_tab3 = ("Filename", "Status", "File_Size_A", "File_Size_B", "File_Modified_A", "File_Modified_B")
        self.result_tree3 = ttk.Treeview(frame, columns=columns_tab3, show="headings", height=20)
        for col in columns_tab3:
            self.result_tree3.heading(col, text=col)
            if col == "Filename":
                self.result_tree3.column(col, width=300)
            elif col in ("File_Modified_A", "File_Modified_B"):
                self.result_tree3.column(col, width=180)
            else:
                self.result_tree3.column(col, width=120)
        self.result_tree3.pack(fill='both', expand=True)

    # ----- Helpers -----
    def select_folder(self, var):
        folder_selected = filedialog.askdirectory()
        if folder_selected and os.path.isdir(folder_selected):
            var.set(folder_selected)
        else:
            messagebox.showwarning("Invalid folder", "Please select a valid folder.")

    def compare_import_vs_raw(self):
        import_folder = self.import_path.get()
        raw_other_folder = self.raw_other_path.get()
        if not import_folder or not raw_other_folder:
            messagebox.showwarning("Missing folders", "Please select both folders.")
            return

        import_files = list_sas_files_one_level(import_folder)

        raw_dict = list_all_files(
            raw_other_folder,
            recursive=False,
            extensions={".sas7bdat"},
            ignore_hidden=True
        )
        raw_files = {k: v["mtime"] for k, v in raw_dict.items()}

        if not import_files:
            messagebox.showinfo("No files found in Import", "No .sas7bdat files found one level below the selected subfolders.")
```

```
        if not raw_files:
            messagebox.showinfo("No files found in Raw_Other", "No .sas7bdat files found at the root of the selected flat folder.")

        result = compare_sub_vs_flat_display(import_files, raw_files)
        self.show_result(self.result_tree1, result)

    def compare_any_files_top_level(self):
        a_folder = self.anyA_path.get()
        b_folder = self.anyB_path.get()
        if not a_folder or not b_folder:
            messagebox.showwarning("Missing folders", "Please select both folders.")
            return

        extensions = parse_extensions_filter(self.any_ext_filter.get())
        dictA = list_all_files(a_folder, recursive=False, extensions=extensions, ignore_hidden=self.any_ignore_hidden.get())
        dictB = list_all_files(b_folder, recursive=False, extensions=extensions, ignore_hidden=self.any_ignore_hidden.get())
        rows = compare_all_files_top_level(dictA, dictB, threshold_seconds=1)
        self.show_result(self.result_tree3, rows)

    def show_result(self, treeview, results):
        for row in treeview.get_children():
            treeview.delete(row)
        for *values, color in results:
            treeview.insert("", "end", values=values, tags=(color,))
        treeview.tag_configure("red", foreground="red")
        treeview.tag_configure("orange", foreground="orange")
        treeview.tag_configure("green", foreground="green")

    def save_results_to_excel(self):
        data_tab1 = [self.result_tree1.item(row)['values'] for row in self.result_tree1.get_children()]
        data_tab3 = [self.result_tree3.item(row)['values'] for row in self.result_tree3.get_children()]

        if not (data_tab1 or data_tab3):
            messagebox.showinfo("No data", "No results to save. Please run a comparison first.")
            return

        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
        save_path = filedialog.asksaveasfilename(
            defaultextension=".xlsx",
            filetypes=[("Excel files", "*.xlsx")],
            title="Save Comparison Report",
            initialfile=f"comparison_results_{timestamp}.xlsx"
        )
        if not save_path:
            messagebox.showinfo("Cancelled", "Save operation was cancelled.")
            return

        with pd.ExcelWriter(save_path, engine="openpyxl") as writer:
            if data_tab1:
                df1 = pd.DataFrame(data_tab1, columns=["Filename", "Exists in Flat folder", "Diff. in Timestamp"])
                df1.to_excel(writer, sheet_name="Sub_vs_Flat", index=False, startrow=5)
            if data_tab3:
                df3 = pd.DataFrame(data_tab3, columns=["Filename", "Status", "File_Size_A", "File_Size_B", "File_Modified_A",
"File_Modified_B"])
                df3.to_excel(writer, sheet_name="Compare_AllFiles_Top", index=False, startrow=12)

        wb = load_workbook(save_path)

        def set_column_width(ws, width=30):
            for col in ws.columns:
                try:
                    ws.column_dimensions[col[0].column_letter].width = width
                except Exception:
                    pass

        if "Sub_vs_Flat" in wb.sheetnames:
            ws1 = wb["Sub_vs_Flat"]
            ws1["A1"] = f"subfolders: {self.import_path.get()}"
            ws1["A2"] = f"flat folder: {self.raw_other_path.get()}"
            ws1["A3"] = f"Total Files Compared: {len(data_tab1)}"
            ws1["A4"] = f"Files with Changed Timestamp: {sum(1 for r in data_tab1 if r[2] == 'Yes')}"
            for cell in ["A1", "A2", "A3", "A4"]:
```

```
            ws1[cell].font = Font(bold=True)
        set_column_width(ws1)

    if "Compare_AllFiles_Top" in wb.sheetnames:
        ws3 = wb["Compare_AllFiles_Top"]
        ws3["A1"] = f"Folder A: {self.anyA_path.get()}"
        ws3["A2"] = f"Folder B: {self.anyB_path.get()}"
        ws3["A3"] = f"Scope: Top-Level files only"
        ws3["A4"] = f"Ignore hidden: {self.any_ignore_hidden.get()}"
        ws3["A5"] = f"Extension filter: {self.any_ext_filter.get()}"
        ws3["A6"] = f"Total Rows: {len(data_tab3)}"
        onlyA = sum(1 for r in data_tab3 if r[1] == "Only in A")
        onlyB = sum(1 for r in data_tab3 if r[1] == "Only in B")
        changed = sum(1 for r in data_tab3 if r[1] == "Changed")
        same = sum(1 for r in data_tab3 if r[1] == "Same")
        ws3["A7"] = f"Only in A: {onlyA}"
        ws3["A8"] = f"Only in B: {onlyB}"
        ws3["A9"] = f"Changed: {changed}"
        ws3["A10"] = f"Same: {same}"
        for cell in ["A1","A2","A3","A4","A5","A6","A7","A8","A9","A10"]:
            ws3[cell].font = Font(bold=True)
        set_column_width(ws3)

    wb.save(save_path)
    messagebox.showinfo("Success", f"Report saved to:\n{save_path}")

# ---------- Launch ----------
def launch_app():
    try:
        root = tk.Tk()
        app = FolderComparerApp(root)
        root.mainloop()
    except Exception as e:
        print("Application failed to start:", e)
        try:
            messagebox.showerror("Startup Error", f"Application failed to start:\n{e}")
        except Exception:
            pass

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