UW Automation Program

Complete Project Documentation

Generated on: July 11, 2025 at 11:35 AM

Project Location: C:\Users\DamionMorrison\OneDrive - True Rx Health Strategists\UW Automation Program

Table of Contents

Section	Description
1. Project Overview	High-level project structure and purpose
2. Configuration Files	JSON configs and settings
3. Main Application	Core application files (app.py)
4. Modules	Individual processing modules
5. Utilities	Helper functions and utilities
6. User Interface	UI components and builders
7. Tests	Test files and test configurations
8. Resources	Documentation and guides
9. File Structure	Complete directory tree

1. Project Overview

The UW Automation Program is a comprehensive Python-based application designed for healthcare repricing automation. It provides tools for data processing, file merging, template operations, and audit logging.

Metric	Count
Total Files	78
Python Files	52
Configuration Files	12
Test Files	2

2. Configuration Files

Configuration files that control application behavior

■ *config__init__.py*

Type: Python Module

■ *config\app_config.py*

Type: Python Module

```
"""
Configuration module for the Repricing Automation application.
Contains configuration classes and constants.
"""

import multiprocessing
import json
import os
from pathlib import Path


class ProcessingConfig:
    """Configuration class for processing settings and validation."""

    REQUIRED_COLUMNS = [
        "DATEFILLED", "SOURCERECORDID", "QUANTITY", "DAYSUPPLY", "NDC",
        "MemberID", "Drug Name", "Pharmacy Name", "Total AWP (Historical)"
    ]

    FILE_TYPES = [
        ("All files", "*.*"),
        ("CSV files", "*.csv"),
        ("Excel files", "*.xlsx"),
    ]

    TEMPLATE_FILE_TYPES = [("Excel files", "*.xlsx")]

    DEFAULT_OPPORTUNITY_NAME = "claims detail PCU"

    @classmethod
    def get_multiprocessing_workers(cls):
        """Get the optimal number of workers for multiprocessing."""
        return min(4, max(1, multiprocessing.cpu_count() // 2))

    @classmethod
    def validate_required_columns(cls, df):
        """Validate that all required columns are present in the DataFrame."""
        missing_cols = [col for col in cls.REQUIRED_COLUMNS if col not in df.columns]
        if missing_cols:
            raise ValueError(f"Missing required columns: {missing_cols}")
        return True


class DisruptionConfig:
    """Configuration for disruption types to reduce conditional complexity."""

    DISRUPTION_TYPES = {
        "Tier Disruption": {
            "module": "modules.tier_disruption",
            "file": "tier_disruption.py"
        },
        "B/G Disruption": {
            "module": "modules.bg_disruption",
            "file": "bg_disruption.py"
        },
        "OpenMDF (Tier)": {
```

```
            "module": "modules.openmdf_tier",
            "file": "openmdf_tier.py"
        },
        "OpenMDF (B/G)": {
            "module": "modules.openmdf_bg",
            "file": "openmdf_bg.py"
        },
        "Full Claims File": {
            "module": "modules.full_claims",
            "file": "full_claims.py"
        }
    }

    @classmethod
    def get_program_file(cls, disruption_type):
        """Get the program file for a disruption type."""
        config = cls.DISRUPTION_TYPES.get(disruption_type)
        return config["file"] if config else None

    @classmethod
    def get_disruption_labels(cls):
        """Get list of available disruption types (excluding Full Claims File)."""
        return [
            label for label in cls.DISRUPTION_TYPES.keys()
            if label != "Full Claims File"
        ]


class AppConstants:
    """Application constants and configuration values."""

    # Configuration and audit log files
    CONFIG_FILE = Path("config.json")
    AUDIT_LOG = Path("audit_log.csv")  # Default fallback

    @classmethod
    def get_audit_log_path(cls):
        """Get the audit log path from configuration."""
        try:
            config_path = Path(__file__).parent / "file_paths.json"
            with open(config_path, 'r') as f:


... (content truncated) ...
```

■ *config\config.json*

Type: Configuration File

```
{
    "last_folder": "C:\\Users\\DamionMorrison\\OneDrive - True Rx Health Strategists\\Python Editor Envior
}
```

■ *config\config_loader.py*

Type: Python Module

■ *config\file_paths.json*

Type: Configuration File

```
{
  "medi_span": "%OneDrive%/True Community - Data Analyst/Medispan Export 1.9.25.xlsx",
  "e_disrupt": "%OneDrive%/True Community - Data Analyst/Repricing Templates/Disruption/Formulary Tiers Re
  "u_disrupt": "%OneDrive%/True Community - Data Analyst/Repricing Templates/Disruption/Formulary Tiers Re
  "mdf_disrupt": "%OneDrive%/True Community - Data Analyst/Repricing Templates/Disruption/Formulary Tiers
```

```
  "n_disrupt": "%OneDrive%/True Community - Data Analyst/Network Check 1.24/Rx Sense Pharmacy 8.23.xlsx",
  "reprice": "_Rx Repricing_wf.xlsx",
  "sharx": "%OneDrive%/True Community - Data Analyst/Repricing Templates/SHARx/Template_Rx Claims for SHAR
  "epls": "%OneDrive%/True Community - Data Analyst/Repricing Templates/EPLS/Client Name_Rx Claims for EPL
  "pharmacy_validation": "%OneDrive%/True Community - Data Analyst/UW Python Program/Logs/Pharmacy_RxSense
  "audit_log": "%OneDrive%/True Community - Data Analyst/UW Python Program/Logs/Shared_Log.csv"
}
```

config\improved_config_manager.py

Type: Python Module

```
"""
Configuration Management - CodeScene ACE Improvement
Centralized configuration handling with better error management
"""

import json
import logging
from pathlib import Path
from typing import Dict, Any, Optional
from dataclasses import dataclass, asdict

logger = logging.getLogger(__name__)


@dataclass
class AppSettings:
    """Application settings data class."""
    last_folder: str
    theme: str = "light"
    auto_save: bool = True
    log_level: str = "INFO"
    max_workers: int = 4
    backup_enabled: bool = True

    @classmethod
    def default(cls) -> 'AppSettings':
        """Create default settings."""
        return cls(
            last_folder=str(Path.cwd()),
            theme="light",
            auto_save=True,
            log_level="INFO",
            max_workers=4,
            backup_enabled=True
        )


class ConfigurationManager:
    """
    Improved configuration management following CodeScene ACE principles.
    - Single responsibility: Only handles configuration
    - Better error handling
    - Type safety with dataclasses
    - Clear separation of concerns
    """

    def __init__(self, config_file: Optional[Path] = None):
        self.config_file = config_file or Path("config.json")
        self._settings: Optional[AppSettings] = None
        self._load_configuration()

    def _load_configuration(self) -> None:
        """Load configuration from file or create default."""
        try:
            if self.config_file.exists():
                self._settings = self._load_from_file()
                logger.info(f"Configuration loaded from {self.config_file}")
            else:
                self._settings = AppSettings.default()
                self._save_configuration()
                logger.info("Default configuration created")
        except Exception as e:
```

```
            logger.error(f"Failed to load configuration: {e}")
            self._settings = AppSettings.default()

    def _load_from_file(self) -> AppSettings:
        """Load settings from JSON file."""
        try:
            with open(self.config_file, 'r') as f:
                data = json.load(f)
                return AppSettings(**data)
        except (json.JSONDecodeError, TypeError) as e:
            logger.warning(f"Invalid configuration file: {e}")
            return AppSettings.default()

    def _save_configuration(self) -> None:
        """Save current settings to file."""
        try:
            if self._settings is not None:
                with open(self.config_file, 'w') as f:
                    json.dump(asdict(self._settings), f, indent=4)
                logger.debug(f"Configuration saved to {self.config_file}")
            else:
                logger.error("No settings to save: self._settings is None")
                return
        except Exception as e:
            logger.error(f"Failed to save configuration: {e}")

    @property
    def settings(self) -> AppSettings:
        """Get current settings.""

... (content truncated) ...
```

3. Main Application

Core application entry points and main logic

■ *app.py*

Type: Python Module

```
import tkinter as tk
import customtkinter as ctk
from tkinter import filedialog, messagebox, scrolledtext
import subprocess
import os
from utils.utils import write_shared_log
from modules.audit_helper import log_file_access, log_process_action, log_system_error, log_file_error
import logging
import threading
import multiprocessing
import pandas as pd
from typing import Optional
import numpy as np
from openpyxl import load_workbook
from openpyxl.styles import PatternFill
from pathlib import Path
import json
import time
import re
import importlib
import importlib.util
import warnings

# Import custom modules
# Theme colors now handled by UIBuilder
from config.app_config import ProcessingConfig, AppConstants
from modules.file_processor import FileProcessor
from modules.template_processor import TemplateProcessor
from modules.data_processor import DataProcessor
from modules.process_manager import ProcessManager
from modules.ui_builder import UIBuilder
from modules.log_manager import LogManager, ThemeController

# Excel COM check
XLWINGS_AVAILABLE = importlib.util.find_spec("xlwings") is not None
EXCEL_COM_AVAILABLE = importlib.util.find_spec("win32com.client") is not None

# Logging setup
logging.basicConfig(
    filename="repricing_log.log",
    level=logging.INFO,
    format="%(asctime)s - %(levelname)s - %(message)s",
    datefmt="%Y-%m-%d %H:%M:%S",
)
logger = logging.getLogger(__name__)


class ConfigManager:
    def __init__(self):
        self.config = {}
        if AppConstants.CONFIG_FILE.exists():
            self.load()
        else:
            self.save_default()

    def save_default(self):
        self.config = {"last_folder": str(Path.cwd())}
        self.save()

    def load(self):
        with open(AppConstants.CONFIG_FILE, "r") as f:
            self.config = json.load(f)

    def save(self):
        with open(AppConstants.CONFIG_FILE, "w") as f:
```

```
            json.dump(self.config, f, indent=4)


class App:
    def __init__(self, root):
        self.root = root
        self._initialize_variables()
        self._initialize_processors()
        self.ui_builder.build_complete_ui()
        self.theme_controller.apply_initial_theme()
        self.log_manager.initialize_logging()
        self.log_manager.log_application_start()

        # Set up proper window close handler for audit logging
        self.root.protocol("WM_DELETE_WINDOW", self.on_closing)

    def on_closing(self):
        """Handle application closing with proper audit logging."""
        try:
            self.log_manager.log_application_shutdown()
        except Exception as e:
            logging.error(f"Failed to log shutdown: {e}")
        finally:
            self.root.destroy()

    def _initialize_variables(self):
        """Initialize all instance variables."""
        self.file1_path = None
        self.file2_path = None
        self.template_file_path = None
        self.cancel_event = threading.Event()
        self.start_time = None
        self.progress

... (content truncated) ...
```

4. Modules

Individual processing modules and components

■ *modules__init__.py*

Type: Python Module

■ *modules\audit_helper.py*

Type: Python Module

```
import logging
import os
import sys
import getpass
import platform
import socket
from datetime import datetime
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from utils.utils import write_shared_log


def make_audit_entry(script_name, message, status="INFO"):
    """Enhanced audit entry with better error handling and user tracking."""
    try:
        write_shared_log(script_name, message, status)
    except Exception as e:
        logging.error(f"[AUDIT FAIL] {script_name} audit failed: {e}")
        try:
            with open("local_fallback_log.txt", "a") as f:
                f.write(f"{script_name}: {message} [{status}]\n")
        except Exception as inner:
            logging.error(f"[FALLBACK FAIL] Could not write fallback log: {inner}")


def log_user_session_start(script_name="Application"):
    """Log comprehensive user session start information."""
    try:
        username = getpass.getuser()
        computer_name = socket.gethostname()
        python_version = platform.python_version()
        os_info = f"{platform.system()} {platform.release()}"

        session_info = (
            f"User: {username} | Computer: {computer_name} | "
            f"Python: {python_version} | OS: {os_info}"
        )

        make_audit_entry(script_name, f"Session started - {session_info}", "START")
        logging.info(f"Session started for user: {username}")

    except Exception as e:
        logging.error(f"Failed to log session start: {e}")
        make_audit_entry(script_name, f"Session start logging failed: {e}", "ERROR")


def log_user_session_end(script_name="Application"):
    """Log user session end information."""
    try:
        username = getpass.getuser()
        computer_name = socket.gethostname()

        session_info = f"User: {username} | Computer: {computer_name}"
        make_audit_entry(script_name, f"Session ended - {session_info}", "END")
        logging.info(f"Session ended for user: {username}")

    except Exception as e:
```

```
        logging.error(f"Failed to log session end: {e}")
        make_audit_entry(script_name, f"Session end logging failed: {e}", "ERROR")


def log_file_access(script_name, file_path, action="ACCESS"):
    """Log file access with user information."""
    try:
        username = getpass.getuser()
        make_audit_entry(script_name, f"File {action}: {file_path} by {username}", "FILE_ACCESS")

    except Exception as e:
        logging.error(f"Failed to log file access: {e}")
        make_audit_entry(script_name, f"File access logging failed: {e}", "ERROR")


def log_process_action(script_name, action, details=""):
    """Log process actions with user information."""
    try:
        username = getpass.getuser()
        message = f"Process {action} by {username}"
        if details:
            message += f" - {details}"
        make_audit_entry(script_name, message, "PROCESS")

    except Exception as e:
        loggi

... (content truncated) ...
```

■ *modules\bg_disruption.py*

Type: Python Module

```
import pandas as pd
import logging
import os
import sys
from pathlib import Path
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from utils.utils import (
    load_file_paths,
    filter_logic_and_maintenance,
    filter_products_and_alternative,
    standardize_pharmacy_ids,
    standardize_network_ids,
    merge_with_network,
    drop_duplicates_df,
    clean_logic_and_tier,
    filter_recent_date,
    write_shared_log,
)
from modules.audit_helper import (
    make_audit_entry,
    log_user_session_start,
    log_user_session_end,
    log_file_access,
)

# Logging setup
logging.basicConfig(
    filename="bg_disruption.log",
    level=logging.INFO,
    format="%(asctime)s - %(levelname)s - %(message)s",
    datefmt="%Y-%m-%d %H:%M:%S",
)
logger = logging.getLogger(__name__)


def load_data_files(file_paths):
    """Load and return all required data files."""
    logger.info("Loading data files...")

    # Load claims data
    try:
```

```
            claims = pd.read_excel(
                file_paths["reprice"],
                sheet_name="Claims Table",
                usecols=[
                    "SOURCERECORDID",
                    "NDC",
                    "MemberID",
                    "DATEFILLED",
                    "FormularyTier",
                    "Rxs",
                    "Logic",
                    "PHARMACYNPI",
                    "NABP",
                    "Pharmacy Name",
                    "Universal Rebates",
                    "Exclusive Rebates",
                ],
            )
        except Exception as e:
            logger.warning(f"Claims Table fallback: {e}")
            make_audit_entry("bg_disruption.py", f"Claims Table fallback error: {e}", "FILE_ERROR")
            write_shared_log(
                "bg_disruption.py", f"Claims Table fallback: {e}", status="WARNING"
            )
            claims = pd.read_excel(file_paths["reprice"], sheet_name=0)

        logger.info(f"claims shape: {claims.shape}")
        claims.info()

        # Load other data files
        medi = pd.read_excel(file_paths["medi_span"])[
            ["NDC", "Maint Drug?", "Product Name"]
        ]
        logger.info(f"medi shape: {medi.shape}")

        uni = pd.read_excel(file_paths["u_disrupt"], sheet_name="Universal NDC")[
            ["NDC", "Tier"]
        ]
        logger.info(f"uni shape: {uni.shape}")

        exl = pd.read_excel(file_paths["e_disrupt"], sheet_name="Alternatives NDC")[
            ["NDC", "Tier", "Alternative"]
        ]
        logger.info(f"exl shape: {exl.shape}")

        network = pd.read_excel(file_paths["n_disrupt"])[
            ["pharmacy_npi", "pharmacy_nabp", "pharmacy_is_excluded"]
        ]
        logger.info(f"network shape: {network.shape}")

        return claims, medi, uni, exl, network


    def merge_data_files(claims, reference_data, network):
        """Merge all data files into a single DataFrame."""
        logger.info("Merging data files...")

        medi, uni, exl = reference_data

        df = claims.merge(medi, on="NDC", how="left")
        logger.info(f"After merge with medi: {df.shape}")

        df = df.merge(uni.rename(c

    ... (content truncated) ...
```

■ *modules\check_audit.py*

Type: Python Module

```
import os
import csv
import json
from pathlib import Path
```

```
# Load the audit log path from config
config_path = Path(__file__).parent.parent / "config" / "file_paths.json"
with open(config_path, 'r') as f:
    file_paths = json.load(f)
log_path = os.path.expandvars(file_paths["audit_log"])

print(f"Audit log path: {log_path}")
print(f"File exists: {os.path.exists(log_path)}")

if os.path.exists(log_path):
    print(f"File size: {os.path.getsize(log_path)} bytes")

    try:
        with open(log_path, 'r', encoding='utf-8', newline='') as f:
            reader = csv.reader(f)
            rows = list(reader)

        print(f"Total rows: {len(rows)}")

        if len(rows) > 0:
            print(f"Header: {rows[0]}")

        if len(rows) > 5:
            print("\nLast 5 entries:")
            for i, row in enumerate(rows[-5:], start=len(rows)-4):
                print(f"Row {i}: {row}")
        else:
            print("\nAll entries:")
            for i, row in enumerate(rows, start=1):
                print(f"Row {i}: {row}")

    except Exception as e:
        print(f"Error reading CSV: {e}")

        # Try reading as text
        try:
            with open(log_path, 'r', encoding='utf-8') as f:
                content = f.read()
            print(f"Raw content (last 500 chars):\n{content[-500:]}")
        except Exception as e2:
            print(f"Error reading as text: {e2}")
```

■ *modules\data_processor.py*

Type: Python Module

```
"""
Data processing module for handling CSV/Excel data operations.
Extracted from app.py to improve cohesion and reduce file size.
"""

import pandas as pd
import numpy as np
import logging
import multiprocessing
from pathlib import Path
import re
import os
import sys

from config.app_config import ProcessingConfig
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from utils.utils import write_shared_log


class DataProcessor:
    """Handles data processing and validation operations."""

    def __init__(self, app_instance):
        self.app = app_instance

    def load_and_validate_data(self, file_path):
        """Load and validate the merged file data."""
        try:
```

```
            df = pd.read_excel(file_path)
            logging.info(f"Loaded {len(df)} records from {file_path}")

            # Validate required columns using configuration
            ProcessingConfig.validate_required_columns(df)

            # Prepare data for processing
            df = df.sort_values(by=["DATEFILLED", "SOURCERECORDID"], ascending=True)
            df["Logic"] = ""
            df["RowID"] = np.arange(len(df))

            return df

        except Exception as e:
            error_msg = f"Error loading data from {file_path}: {str(e)}"
            logging.error(error_msg)
            write_shared_log("DataProcessor", error_msg, "ERROR")
            raise

    def process_data_multiprocessing(self, df):
        """Process data using multiprocessing for improved performance."""
        try:
            # Import and use multiprocessing helpers
            from modules import mp_helpers

            num_workers = ProcessingConfig.get_multiprocessing_workers()
            df_blocks = np.array_split(df, num_workers)
            out_queue = multiprocessing.Queue()
            processes = []

            # Start worker processes
            for block in df_blocks:
                p = multiprocessing.Process(
                    target=mp_helpers.worker, args=(block, out_queue)
                )
                p.start()
                processes.append(p)

            # Collect results
            results = [out_queue.get() for _ in processes]
            for p in processes:
                p.join()

            processed_df = pd.concat(results)
            logging.info(f"Processed {len(processed_df)} records using {num_workers} workers")
            return processed_df

        except Exception as e:
            error_msg = f"Error processing data with multiprocessing: {str(e)}"
            logging.error(error_msg)
            write_shared_log("DataProcessor", error_msg, "ERROR")
            raise

    def save_processed_outputs(self, df, output_dir=None):
        """Save processed data to various output formats."""
        try:
            if output_dir is None:
                output_dir = Path.cwd()


... (content truncated) ...
```

■ *modules\epls_lbl.py*

Type: Python Module

```
import logging
import tkinter as tk
from pathlib import Path
from tkinter import messagebox
import os
import sys

import pandas as pd
```

```
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from utils.excel_utils import write_df_to_template
from utils.utils import load_file_paths, write_shared_log
from modules.audit_helper import (
    make_audit_entry,
    log_user_session_start,
    log_user_session_end,
    log_file_access,
)

# Setup logging
logging.basicConfig(
    filename="epls_lbl.log",
    level=logging.INFO,
    format="%(asctime)s - %(levelname)s - %(message)s",
    datefmt="%Y-%m-%d %H:%M:%S",
)
logger = logging.getLogger(__name__)


def show_message(awp, ing, total, rxs):
    messagebox.showinfo(
        "Process Complete",
        f"EPLS LBL has been created\n\n"
        f"Total AWP: ${awp:.2f}\n\n"
        f"Total Ing Cost: ${ing:.2f}\n\n"
        f"Total Gross Cost: ${total:.2f}\n\n"
        f"Total Claim Count: {rxs}",
    )


def main() -> None:
    tk.Tk().withdraw()

    # Start audit session
    log_user_session_start("epls_lbl.py")
    write_shared_log("epls_lbl.py", "Processing started.")

    try:
        # Get the config file path relative to the project root
        config_path = Path(__file__).parent.parent / "config" / "file_paths.json"
        paths = load_file_paths(str(config_path))

        # Failsafe: check that both input and template files exist
        for key in ["reprice", "epls"]:
            if not Path(paths[key]).exists():
                make_audit_entry("epls_lbl.py", f"{key} path not found: {paths[key]}", "FILE_ERROR")
                raise FileNotFoundError(f"{key} path not found: {paths[key]}")

        template_path = Path(paths["epls"])

        # Log file access
        log_file_access("epls_lbl.py", paths["reprice"], "LOADING")
        log_file_access("epls_lbl.py", paths["epls"], "LOADING")

        df = pd.read_excel(paths["reprice"], sheet_name="Claims Table")

        # Debug print and log
        print("Columns in claims DataFrame:")
        print(df.columns)
        logger.info(f"Columns in claims DataFrame: {df.columns.tolist()}")

        if "Logic" not in df.columns:
            make_audit_entry("epls_lbl.py", "Missing 'Logic' column in Claims Table", "DATA_ERROR")
            raise KeyError("Missing 'Logic' column in Claims Table.")

        df["Logic"] = pd.to_numeric(df["Logic"], errors="coerce")
        df = df[df["Logic"].between(1, 10)]

        awp = df["Total AWP (Historical)"].sum()
        ing = df["Rx Sense Ing Cost"].sum()
        total = df["RxSense Total Cost"].sum()
        rxs = df["Rxs"].sum()

        columns_to_keep = [
            "MONY",
            "Rxs",
```

```
            "Rx Sense Ing Cost",
            "RxSense Dispense Fee",
            "RxSense Total Cost",
            "Total AWP (Historical)",
            "Pharmacy Name",
            "INPUTFILECHANNEL",
            "DATEFILLED",
            "MemberID",
            "DAYSUP

... (content truncated) ...
```

■ *modules\error_analysis_tool.py*

Type: Python Module

```
"""
Error Analysis and Support Helper
Provides tools to analyze audit logs and generate support reports for user assistance.
"""

import pandas as pd
import os
import json
from datetime import datetime, timedelta
from pathlib import Path

def get_audit_log_path():
    """Get the path to the audit log file from config."""
    config_path = Path(__file__).parent.parent / "config" / "file_paths.json"
    with open(config_path, 'r') as f:
        file_paths = json.load(f)
    return os.path.expandvars(file_paths["audit_log"])

def get_user_errors(username=None, days_back=7, error_types=None):
    """
    Get errors for a specific user or all users within a date range.

    Args:
        username: Specific username to filter by (None for all users)
        days_back: Number of days to look back (default 7)
        error_types: List of error types to filter by (e.g., ['USER_ERROR', 'SYSTEM_ERROR'])

    Returns:
        DataFrame with filtered error entries
    """
    try:
        log_path = get_audit_log_path()
        if not os.path.exists(log_path):
            print(f"Audit log not found at: {log_path}")
            return pd.DataFrame()

        # Read the CSV file
        df = pd.read_csv(log_path)

        # Convert timestamp to datetime
        df['Timestamp'] = pd.to_datetime(df['Timestamp'])

        # Filter by date range
        cutoff_date = datetime.now() - timedelta(days=days_back)
        df = df[df['Timestamp'] >= cutoff_date]

        # Filter by error types
        if error_types is None:
            error_types = ['USER_ERROR', 'SYSTEM_ERROR', 'FILE_ERROR', 'DATA_ERROR']
        df = df[df['Status'].isin(error_types)]

        # Filter by username if specified
        if username:
            df = df[df['User'].str.contains(username, case=False, na=False)]

        # Sort by timestamp (newest first)
        df = df.sort_values('Timestamp', ascending=False)
```

```
            return df

        except Exception as e:
            print(f"Error reading audit log: {e}")
            return pd.DataFrame()

    def generate_user_support_report(username, days_back=30):
        """
        Generate a comprehensive support report for a specific user.

        Args:
            username: Username to generate report for
            days_back: Number of days to analyze (default 30)

        Returns:
            String containing formatted support report
        """
        try:
            log_path = get_audit_log_path()
            df = pd.read_csv(log_path)
            df['Timestamp'] = pd.to_datetime(df['Timestamp'])

            # Filter for the user and date range
            cutoff_date = datetime.now() - timedelta(days=days_back)
            user_df = df[
                (df['User'].str.contains(username, case=False, na=False)) &
                (df['Timestamp'] >= cutoff_date)
            ]

            # Get error entries
            error_df = user_df[user_df['Status'].isin(['USER_ERROR', 'SYSTEM_ERROR', 'FILE_ERROR', 'DATA_ERROR


    ... (content truncated) ...
```

■ *modules\file_processor.py*

Type: Python Module

```
"""
File processing module for handling file operations and validation.
Extracted from app.py to improve cohesion and reduce file size.
"""

import pandas as pd
from pathlib import Path
import os
import sys
from tkinter import messagebox

from config.app_config import ProcessingConfig, AppConstants
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from utils.utils import write_shared_log


class FileProcessor:
    """Handles file operations and validation."""

    def __init__(self, app_instance):
        self.app = app_instance

    def check_template(self, file_path):
        """Check if template file exists and is valid."""
        return Path(file_path).exists() and Path(file_path).suffix == '.xlsx'

    def import_file(self, file_type="File"):
        """Import and validate a file."""
        file_path = self.app.ui_factory.create_file_dialog(
            title=f"Select {file_type}",
            filetypes=ProcessingConfig.FILE_TYPES
        )

        if not file_path:
```

```
            return None

        try:
            # Validate file exists
            if not Path(file_path).exists():
                messagebox.showerror("Error", f"{file_type} not found.")
                return None

            # Load and validate the file
            df = pd.read_csv(file_path) if file_path.endswith('.csv') else pd.read_excel(file_path)

            # Log the import
            write_shared_log("FileProcessor", f"{file_type} imported successfully: {file_path}")

            return file_path, df

        except Exception as e:
            error_msg = f"Error importing {file_type}: {str(e)}"
            messagebox.showerror("Error", error_msg)
            write_shared_log("FileProcessor", error_msg, "ERROR")
            return None

    def validate_file_structure(self, df, required_columns=None):
        """Validate that the file has the required structure."""
        if required_columns is None:
            required_columns = ProcessingConfig.REQUIRED_COLUMNS

        try:
            ProcessingConfig.validate_required_columns(df)
            return True
        except ValueError as e:
            messagebox.showerror("Validation Error", str(e))
            return False

    def prepare_file_paths(self, template_path, opportunity_name=None):
        """Prepare file paths for template operations."""
        if not template_path:
            raise ValueError("Template file path is not set.")

        template = Path(template_path)
        backup_name = template.stem + AppConstants.BACKUP_SUFFIX

        # Use opportunity name in output filename if provided
        if opportunity_name:
            output_name = f"{opportunity_name}_Rx Repricing_wf.xlsx"
        else:
            output_name = AppConstants.UPDATED_TEMPLATE_NAME

        return {
            "template": template,
            "backup"

... (content truncated) ...
```

■ *modules\log_manager.py*

Type: Python Module

```
"""
Log management module for handling various logging and viewer operations.
Extracted from app.py to reduce file size and improve organization.
"""

import tkinter as tk
from tkinter import scrolledtext
import csv
import os
import sys
import logging
import json
from pathlib import Path

sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from utils.utils import write_shared_log
```

```
from modules.audit_helper import log_user_session_start, log_user_session_end, validate_user_access


class LogManager:
    """Handles log viewing and management operations."""

    def __init__(self, app_instance):
        self.app = app_instance
        # Load the audit log path from config
        config_path = Path(__file__).parent.parent / "config" / "file_paths.json"
        with open(config_path, 'r') as f:
            file_paths = json.load(f)
        self.shared_log_path = os.path.expandvars(file_paths["audit_log"])

    def show_log_viewer(self):
        """Show the live log viewer window."""
        log_win = tk.Toplevel(self.app.root)
        log_win.title("Live Log Viewer")
        text_area = scrolledtext.ScrolledText(log_win, width=100, height=30)
        text_area.pack(fill="both", expand=True)

        def update_logs():
            try:
                with open("repricing_log.log", "r") as f:
                    text_area.delete(1.0, tk.END)
                    text_area.insert(tk.END, f.read())
            except FileNotFoundError:
                text_area.delete(1.0, tk.END)
                text_area.insert(tk.END, "No log file found.")
            except Exception as e:
                text_area.delete(1.0, tk.END)
                text_area.insert(tk.END, f"Error reading log file: {e}")
            log_win.after(3000, update_logs)

        update_logs()

    def show_shared_log_viewer(self):
        """Show the shared audit log viewer with search functionality."""
        log_win = tk.Toplevel(self.app.root)
        log_win.title("Shared Audit Log Viewer")
        log_win.geometry("1000x600")

        # Create filter frame
        filter_frame = tk.Frame(log_win)
        filter_frame.pack(fill="x")
        tk.Label(filter_frame, text="Search:").pack(side="left", padx=5)
        filter_entry = tk.Entry(filter_frame)
        filter_entry.pack(side="left", fill="x", expand=True, padx=5)

        # Create text area
        text_area = scrolledtext.ScrolledText(log_win, width=150, height=30)
        text_area.pack(fill="both", expand=True)

        def refresh():
            """Refresh the log display with optional filtering."""
            try:
                if not os.path.exists(self.shared_log_path):
                    text_area.delete(1.0, tk.END)
                    text_area.insert(tk.END, f"Shared log file not found at: {self.shared_log_path}")
                    return

                with open(self.shared_log_path, "r", newline="", encoding="utf-8") as f:


... (content truncated) ...
```

■ *modules\merge.py*

Type: Python Module

```
import sys
import pandas as pd
from pathlib import Path
from openpyxl import load_workbook
```

```
from openpyxl.styles import NamedStyle
import logging
import os
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from utils.utils import write_shared_log

# Configure logging
logging.basicConfig(
    filename="merge_log.log",
    level=logging.INFO,
    format="%(asctime)s - %(levelname)s - %(message)s",
    datefmt="%Y-%m-%d %H:%M:%S",
)
logger = logging.getLogger(__name__)

MERGED_FILENAME = "merged_file.xlsx"
REQUIRED_COLUMNS = [
    "DATEFILLED",
    "SOURCERECORDID",
    "QUANTITY",
    "DAYSUPPLY",
    "NDC",
    "MemberID",
    "Drug Name",
    "Pharmacy Name",
    "Total AWP (Historical)",
]


def merge_files(file1_path, file2_path):
    file1 = Path(file1_path)
    file2 = Path(file2_path)
    try:
        logger.info(f"Starting merge: {file1} + {file2}")
        write_shared_log("merge.py", f"Starting merge: {file1} + {file2}")

        if not file1.exists():
            logger.error(f"File not found: {file1}")
            write_shared_log("merge.py", f"File not found: {file1}", status="ERROR")
            return False
        if not file2.exists():
            logger.error(f"File not found: {file2}")
            write_shared_log("merge.py", f"File not found: {file2}", status="ERROR")
            return False

        # Load data (support Excel or CSV for both files)
        try:
            if file1.suffix == ".csv":
                df1 = pd.read_csv(file1, parse_dates=["DATEFILLED"], dayfirst=False)
            else:
                df1 = pd.read_excel(file1, parse_dates=["DATEFILLED"])
        except Exception as e:
            logger.error(f"Failed to load file1: {e}")
            write_shared_log("merge.py", f"Failed to load file1: {e}", status="ERROR")
            return False
        try:
            if file2.suffix == ".csv":
                df2 = pd.read_csv(file2)
            else:
                df2 = pd.read_excel(file2)
        except Exception as e:
            logger.error(f"Failed to load file2: {e}")
            write_shared_log("merge.py", f"Failed to load file2: {e}", status="ERROR")
            return False

        # Log data source details
        logger.info(f"df1 shape: {df1.shape}, df2 shape: {df2.shape}")
        write_shared_log("merge.py", f"df1 shape: {df1.shape}, df2 shape: {df2.shape}")
        logger.info(f"df1 columns: {list(df1.columns)}")
        logger.info(f"df2 columns: {list(df2.columns)}")

        # Clean up and standardize column names
        df2.columns = [col.strip() for col in df2.columns]
        if "Source Record ID" in df2.columns:
            df2.rename(columns={"Source Record ID": "SOURCERECORDID"}, inplace=True)

        # Merge
```

```
        try:
            df_merged = pd.merge(df1, df2, on="SOURCERECORDID", how="outer")
        except Exception as e:
            logger.error(f"Failed to merge: {e}")
            write_shared_log("merge.py", f"Failed to merge: {e}", status

... (content truncated) ...
```

■ modules\merge_log.log

Type: .LOG File

Binary or special file - content not displayed

■ modules\merged_file.xlsx

Type: .XLSX File

Binary or special file - content not displayed

■ modules\mp_helpers.py

Type: Python Module

```
import pandas as pd
import numpy as np


def process_logic_block(df_block):
    """
    Vectorized numpy logic to mark 'OR' in 'Logic' for reversals with matching claims.
    Refactored to reduce nesting complexity and improve readability.
    """
    arr = df_block.to_numpy()
    col_idx = {col: i for i, col in enumerate(df_block.columns)}

    # Extract and prepare data
    logic_data = _extract_logic_data(arr, col_idx)

    # Early return if no reversals to process
    if not np.any(logic_data["is_reversal"]):
        return pd.DataFrame(arr, columns=df_block.columns)

    # Process reversals with reduced nesting
    _process_reversals(arr, col_idx, logic_data)

    return pd.DataFrame(arr, columns=df_block.columns)


def _extract_logic_data(arr, col_idx):
    """Extract and prepare data for logic processing."""
    qty = arr[:, col_idx["QUANTITY"]].astype(float)
    return {
        "qty": qty,
        "is_reversal": qty < 0,
        "is_claim": qty > 0,
        "ndc": arr[:, col_idx["NDC"]].astype(str),
        "member": arr[:, col_idx["MemberID"]].astype(str),
        "datefilled": pd.to_datetime(arr[:, col_idx["DATEFILLED"]], errors="coerce"),
        "abs_qty": np.abs(qty)
    }


def _process_reversals(arr, col_idx, logic_data):
    """Process reversals with matching logic, using guard clauses to reduce nesting."""
    rev_idx = np.where(logic_data["is_reversal"])[0]
    claim_idx = (
        np.where(logic_data["is_claim"])[0]
```

```
        if np.any(logic_data["is_claim"])
        else np.array([], dtype=int)
    )

    match_context = {
        "arr": arr,
        "col_idx": col_idx,
        "logic_data": logic_data,
        "claim_idx": claim_idx
    }

    for i in rev_idx:
        found_match = _try_find_match(match_context, i)

        # Mark unmatched reversals as 'OR'
        if not found_match:
            arr[i, col_idx["Logic"]] = "OR"


def _try_find_match(match_context, reversal_idx):
    """Attempt to find a matching claim for a reversal. Returns True if match found."""
    arr = match_context["arr"]
    col_idx = match_context["col_idx"]
    logic_data = match_context["logic_data"]
    claim_idx = match_context["claim_idx"]

    # Guard clause: no claims to match against
    if claim_idx.size == 0:
        return False

    # Find potential matches
    matches = _find_matching_claims(logic_data, claim_idx, reversal_idx)

    # Guard clause: no matches found
    if not np.any(matches):
        return False

    # Mark both reversal and matching claim as 'OR'
    arr[reversal_idx, col_idx["Logic"]] = "OR"
    arr[claim_idx[matches][0], col_idx["Logic"]] = "OR"
    return True


def _find_matching_claims(logic_data, claim_idx, reversal_idx):
    """Find claims that match the reversal based on NDC, member, quantity, and date."""
    matches = (
        (logic_data["ndc"][claim_idx] == logic_data["ndc"][reversal_idx])
        & (logic_data["member"][claim_idx] == logic_data["member"]

... (content truncated) ...
```

■ *modules\openmdf_bg.py*

Type: Python Module

```
import pandas as pd
import logging
import os
import sys
from pathlib import Path
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from utils.utils import (
    load_file_paths,
    standardize_pharmacy_ids,
    standardize_network_ids,
    merge_with_network,
    drop_duplicates_df,
    clean_logic_and_tier,
    filter_recent_date,
    filter_logic_and_maintenance,
    filter_products_and_alternative,
    write_shared_log,
)
from modules.audit_helper import (
```

```
    make_audit_entry,
    log_user_session_start,
    log_user_session_end,
    log_file_access,
)

# Setup logging
logging.basicConfig(
    filename="openmdf_bg.log",
    level=logging.INFO,
    format="%(asctime)s - %(levelname)s - %(message)s",
    datefmt="%Y-%m-%d %H:%M:%S",
)
logger = logging.getLogger(__name__)

# Load NABP/NPI list
included_nabp_npi = {
    "4528874": "1477571404",
    "2365422": "1659313435",
    "3974157": "1972560688",
    "320793": "1164437406",
    "4591055": "1851463087",
    "2348046": "1942303110",
    "4023610": "1407879588",
    "4025385": "1588706212",
    "4025311": "1588705446",
    "4026806": "1285860312",
    "4931350": "1750330775",
    "4024585": "1396768461",
    "4028026": "1497022438",
    "2643749": "1326490376",
}


def process_data():
    # Start audit session
    log_user_session_start("openmdf_bg.py")
    write_shared_log("openmdf_bg.py", "Processing started.")

    try:
        import sys

        # Get the config file path relative to the project root
        config_path = Path(__file__).parent.parent / "config" / "file_paths.json"
        paths = load_file_paths(str(config_path))

        if "reprice" not in paths or not paths["reprice"]:
            logger.warning("No reprice/template file provided.")
            make_audit_entry("openmdf_bg.py", "No reprice/template file provided.", "FILE_ERROR")
            write_shared_log(
                "openmdf_bg.py", "No reprice/template file provided.", status="ERROR"
            )
            print("No reprice/template file provided.")
            log_user_session_end("openmdf_bg.py")
            return False

        # Log file access
        log_file_access("openmdf_bg.py", paths["reprice"], "LOADING")

        # Check for required sheet name in reprice file
        try:
            xl = pd.ExcelFile(paths["reprice"])
            if "Claims Table" not in xl.sheet_names:
                logger.error(
                    f"Sheet 'Claims Table' not found in {paths['reprice']}. Sheets: {xl.sheet_names}"
                )
                make_audit_entry("openmdf_bg.py", f"Sheet 'Claims Table' not found. Available sheets: {xl.
                write_shared_log(
                    "openmdf_bg.py",
                    f"Sheet 'Claims Table' not found in {paths['reprice']}. Sheets: {xl.sheet_names}",
                    status="ERROR",
                )
                log_user_session_end("openmdf_bg.py")
                return False


... (content truncated) ...
```

■ *modules\openmdf_bg_simple.py*

Type: Python Module

... and 9 more files in this category

5. Utilities

Helper functions and utility modules

■ *utils__init__.py*

Type: Python Module

■ *utils\excel_utils.py*

Type: Python Module

```
import os
import shutil
import logging
import xlwings as xw
import importlib.util
from typing import Any, Tuple
import pandas as pd
from concurrent.futures import ThreadPoolExecutor, as_completed

logger = logging.getLogger(__name__)

# COM fallback via pywin32
EXCEL_COM_AVAILABLE = importlib.util.find_spec("win32com.client") is not None


def open_workbook(path: str, visible: bool = False) -> Tuple[Any, Any, bool]:
    """
    Open workbook via xlwings or COM fallback.
    Returns (wb, app_obj, use_com).
    """
    import time

    max_retries = 3
    delay = 2
    last_exc = None
    for attempt in range(max_retries):
        try:
            app = xw.App(visible=visible, add_book=False)  # Ensure no new book is added
            try:
                wb = app.books.open(path)
            except TypeError:
                # Try with password if provided in path (e.g., path='file.xlsx::password')
                if "::" in path:
                    file_path, password = path.split("::", 1)
                    wb = app.books.open(file_path, password=password)
                else:
                    raise
            return wb, app, False
        except Exception as e:
            last_exc = e
            logger.warning(
                f"Failed to open workbook (attempt {attempt + 1}/{max_retries}): {e}"
            )
            time.sleep(delay)
    if EXCEL_COM_AVAILABLE:
        import win32com.client as win32

        excel: Any = win32.Dispatch("Excel.Application")
        excel.Visible = visible  # Ensure Excel remains hidden
        excel.DisplayAlerts = False  # Suppress alerts
        try:
            if "::" in path:
                file_path, password = path.split("::", 1)
                wb: Any = excel.Workbooks.Open(
                    os.path.abspath(file_path), False, False, None, password
                )
```

```
            else:
                wb: Any = excel.Workbooks.Open(os.path.abspath(path))
        except Exception as e:
            logger.error(f"COM fallback failed to open workbook: {e}")
            raise
        return wb, excel, True
    logger.error(f"Failed to open workbook after {max_retries} attempts: {last_exc}")
    if last_exc is not None:
        raise last_exc
    # Should never reach here, but raise as a safeguard
    raise RuntimeError("Failed to open workbook and no exception was captured.")


def write_df_to_sheet_async(
    path: str,
    sheet_name: str,
    df: pd.DataFrame,
    start_cell: str = "A2",
    header: bool = False,
    index: bool = False,
    clear: bool = True,
    visible: bool = False,
    clear_by_label: bool = False,
    max_workers: int = 4,
) -> None:
    """
    Async version of write_df_to_sheet for large DataFrames (xlwings only).
    Splits DataFrame into row blocks and writes in parallel threads.
    """
    logger.info(
        f"[ASYNC] Writing to {path} in sheet '{sheet_name}' from cell {start_cell} with {max_workers} work
    )
    wb, app, use_com = open_workbook(path,

... (content truncated) ...
```

■ *utils\logic_processor.py*

Type: Python Module

```
"""
Logic processing utilities extracted from app.py
Following CodeScene ACE principles for better code organization
"""

import logging
import warnings
from dataclasses import dataclass
from typing import Dict

import numpy as np
import pandas as pd

logger = logging.getLogger(__name__)

# Filter out specific warnings
warnings.filterwarnings("ignore", category=FutureWarning, message=".*swapaxes.*")


@dataclass
class LogicData:
    """Data class to encapsulate logic processing data."""
    qty: np.ndarray
    is_reversal: np.ndarray
    is_claim: np.ndarray
    ndc: np.ndarray
    member: np.ndarray
    datefilled: pd.DatetimeIndex
    abs_qty: np.ndarray


@dataclass
class MatchContext:
    """Context object to encapsulate matching parameters."""
```

```
    arr: np.ndarray
    col_idx: Dict[str, int]
    logic_data: LogicData
    claim_idx: np.ndarray


class LogicProcessor:
    """Handles logic processing for reversal matching."""

    @staticmethod
    def process_logic_block(df_block: pd.DataFrame) -> pd.DataFrame:
        """
        Vectorized numpy logic to mark 'OR' in 'Logic' for reversals with matching claims.
        Refactored to reduce nesting complexity and improve readability.
        """
        arr = df_block.to_numpy()
        col_idx = {col: i for i, col in enumerate(df_block.columns)}

        # Extract and prepare data
        logic_data = LogicProcessor._extract_logic_data(arr, col_idx)

        # Early return if no reversals to process
        if not np.any(logic_data.is_reversal):
            return pd.DataFrame(arr, columns=df_block.columns)

        # Process reversals with reduced nesting
        LogicProcessor._process_reversals(arr, col_idx, logic_data)

        return pd.DataFrame(arr, columns=df_block.columns)

    @staticmethod
    def _extract_logic_data(arr: np.ndarray, col_idx: Dict[str, int]) -> LogicData:
        """Extract and prepare data for logic processing."""
        qty = arr[:, col_idx["QUANTITY"]].astype(float)

        return LogicData(
            qty=qty,
            is_reversal=qty < 0,
            is_claim=qty > 0,
            ndc=arr[:, col_idx["NDC"]].astype(str),
            member=arr[:, col_idx["MemberID"]].astype(str),
            datefilled=pd.to_datetime(arr[:, col_idx["DATEFILLED"]], errors="coerce"),
            abs_qty=np.abs(qty)
        )

    @staticmethod
    def _process_reversals(arr: np.ndarray, col_idx: Dict[str, int], logic_data: LogicData):
        """Process reversals with matching logic, using guard clauses to reduce nesting."""
        rev_idx = np.where(logic_data.is_reversal)[0]
        claim_idx = (
            np.where(logic_data.is_claim)[0]
            if np.any(logic_data.is_claim)
            else np.array([], dtype=int)
        )

        # Create context object to reduce function argument count
        match_context = MatchContext(arr, col_idx, logic_data, claim_idx)

        for i in rev_idx:
            fo

... (content truncated) ...
```

■ *utils\utils.py*

Type: Python Module

```
import pandas as pd
import json
import logging
import os
import csv
from pathlib import Path
import getpass
```

```
from datetime import datetime
from dataclasses import dataclass

# Load the audit log path from config
config_path = Path(__file__).parent.parent / "config" / "file_paths.json"
with open(config_path, 'r') as f:
    file_paths = json.load(f)
shared_log_path = os.path.expandvars(file_paths["audit_log"])


@dataclass
class LogicMaintenanceConfig:
    """Configuration for logic and maintenance filtering."""
    logic_col: str = "Logic"
    min_logic: int = 5
    max_logic: int = 10
    maint_col: str = "Maint Drug?"


def ensure_directory_exists(path):
    """
    Ensures the directory for the given path exists.
    """
    try:
        os.makedirs(os.path.dirname(path), exist_ok=True)
    except Exception as e:
        print(f"[ensure_directory_exists] Error: {e}")


def write_shared_log(script_name, message, status="INFO"):
    """
    Appends a log entry to the shared audit log in OneDrive. Rotates log if too large.
    """
    try:
        username = getpass.getuser()
        timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        log_entry = [timestamp, username, script_name, message, status]

        write_header = not os.path.exists(shared_log_path)
        ensure_directory_exists(shared_log_path)

        # Log rotation: if file > 5MB, rotate (keep 3 backups)
        max_size = 5 * 1024 * 1024
        if (
            os.path.exists(shared_log_path)
            and os.path.getsize(shared_log_path) > max_size
        ):
            for i in range(2, 0, -1):
                prev = f"{shared_log_path}.{i}"
                prev2 = f"{shared_log_path}.{i + 1}"
                if os.path.exists(prev):
                    os.replace(prev, prev2)
            os.replace(shared_log_path, f"{shared_log_path}.1")

        with open(shared_log_path, mode="a", newline="", encoding="utf-8") as file:
            writer = csv.writer(file)
            if write_header:
                writer.writerow(["Timestamp", "User", "Script", "Message", "Status"])
            writer.writerow(log_entry)
    except Exception as e:
        print(f"[Shared Log] Error: {e}")


def log_exception(script_name, exc, status="ERROR"):
    """
    Standardized exception logging to shared log and console.
    """
    import traceback

    tb = traceback.format_exc()
    msg = f"{exc}: {tb}"
    print(f"[Exception] {msg}")
    write_shared_log(script_name, msg, status)


def load_file_paths(json_file="file_paths.json"):
    """
```

```
    Loads a JSON config file, replacing %OneDrive% with the user's OneDrive path.
    Returns a dictionary mapping keys to resolved absolute file paths.
    """
    try:
        with open(json_file, "r") as f:
            paths = json.load(f)

        # Resolve the user's OneDrive path
        onedrive_path = os.environ.get("OneDrive")
        if not onedrive_path:
            raise EnvironmentError(
                "OneDrive environment variable

... (content truncated) ...
```

utils\utils_functions.py

Type: Python Module

```
import pandas as pd
import json
import logging
import os
import csv
from pathlib import Path
import getpass
from datetime import datetime

# Load the audit log path from config
config_path = Path(__file__).parent.parent / "config" / "file_paths.json"
with open(config_path, 'r') as f:
    file_paths = json.load(f)
shared_log_path = os.path.expandvars(file_paths["audit_log"])


def ensure_directory_exists(path):
    """
    Ensures the directory for the given path exists.
    """
    try:
        os.makedirs(os.path.dirname(path), exist_ok=True)
    except Exception as e:
        print(f"[ensure_directory_exists] Error: {e}")


def write_shared_log(script_name, message, status="INFO"):
    """
    Appends a log entry to the shared audit log in OneDrive. Rotates log if too large.
    """
    try:
        username = getpass.getuser()
        timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        log_entry = [timestamp, username, script_name, message, status]

        write_header = not os.path.exists(shared_log_path)
        ensure_directory_exists(shared_log_path)

        # Log rotation: if file > 5MB, rotate (keep 3 backups)
        max_size = 5 * 1024 * 1024
        if (
            os.path.exists(shared_log_path)
            and os.path.getsize(shared_log_path) > max_size
        ):
            for i in range(2, 0, -1):
                prev = f"{shared_log_path}.{i}"
                prev2 = f"{shared_log_path}.{i + 1}"
                if os.path.exists(prev):
                    os.replace(prev, prev2)
            os.replace(shared_log_path, f"{shared_log_path}.1")

        with open(shared_log_path, mode="a", newline="", encoding="utf-8") as file:
            writer = csv.writer(file)
            if write_header:
                writer.writerow(["Timestamp", "User", "Script", "Message", "Status"])
```

```
            writer.writerow(log_entry)
    except Exception as e:
        print(f"[Shared Log] Error: {e}")


def log_exception(script_name, exc, status="ERROR"):
    """
    Standardized exception logging to shared log and console.
    """
    import traceback

    tb = traceback.format_exc()
    msg = f"{exc}: {tb}"
    print(f"[Exception] {msg}")
    write_shared_log(script_name, msg, status)


def load_file_paths(json_file="file_paths.json"):
    """
    Loads a JSON config file, replacing %OneDrive% with the user's OneDrive path.
    Returns a dictionary mapping keys to resolved absolute file paths.
    """
    try:
        with open(json_file, "r") as f:
            paths = json.load(f)

        # Resolve the user's OneDrive path
        onedrive_path = os.environ.get("OneDrive")
        if not onedrive_path:
            raise EnvironmentError(
                "OneDrive environment variable not found. Please ensure OneDrive is set up."
            )

        resolved_paths = {}
        for key, path in paths.items():
            if path.startswith("%OneDrive%"):
                path = path.replace("%OneDrive%", onedrive_path)


... (content truncated) ...
```

6. User Interface

User interface components and builders

ui\ui_components.py

Type: Python Module

```
"""
UI Components module for the Repricing Automation application.
This module contains UI-related classes and utilities to improve code organization.
"""

import customtkinter as ctk


# UI styling variables
FONT_SELECT = ("Cambria", 20, "bold")

# Color palettes
LIGHT_COLORS = {
    "dark_blue": "#D9EAF7",
    "grey_blue": "#A3B9CC",
    "mint": "#8FD9A8",
    "button_red": "#D52B2B",
}

DARK_COLORS = {
    "dark_blue": "#223354",
    "grey_blue": "#31476A",
    "mint": "#26A69A",
    "button_red": "#931D1D",
}


class UIFactory:
    """Factory class to create UI components and reduce code duplication."""

    @staticmethod
    def _create_button_base(parent, text, command, fg_color):
        """Base method for creating buttons with common styling."""
        return ctk.CTkButton(
            parent,
            text=text,
            command=command,
            font=FONT_SELECT,
            height=40,
            fg_color=fg_color,
            text_color="#000000"
        )

    @staticmethod
    def create_standard_button(parent, text, command):
        """Create a standardized button with common styling."""
        return UIFactory._create_button_base(parent, text, command, LIGHT_COLORS["mint"])

    @staticmethod
    def create_red_button(parent, text, command):
        """Create a red button (for cancel/exit actions)."""
        return UIFactory._create_button_base(parent, text, command, LIGHT_COLORS["button_red"])

    @staticmethod
    def create_standard_frame(parent):
        """Create a standardized frame with common styling."""
        return ctk.CTkFrame(parent, fg_color=LIGHT_COLORS["grey_blue"])

    @staticmethod
    def create_standard_label(parent, text, width=None):
        """Create a standardized label."""
        if width:
            return ctk.CTkLabel(parent, text=text, font=FONT_SELECT, width=width)
        return ctk.CTkLabel(parent, text=text, font=FONT_SELECT)
```

```
class ThemeManager:
    """Manages theme colors and application of themes to UI components."""

    @staticmethod
    def apply_theme_colors(app_instance, colors):
        """Apply theme colors to all UI components."""
        ThemeManager._apply_root_colors(app_instance, colors)
        ThemeManager._apply_frame_colors(app_instance, colors)
        ThemeManager._apply_button_colors(app_instance, colors)
        ThemeManager._apply_special_component_colors(app_instance, colors)

    @staticmethod
    def _apply_root_colors(app_instance, colors):
        """Apply colors to the root window."""
        app_instance.root.configure(fg_color=colors["dark_blue"])

    @staticmethod
    def _apply_frame_colors(app_instance, colors):
        """Apply colors to frames."""
        frames = ["button_frame", "notes_frame", "dis_frame", "prog_frame"]
        for frame_name in frames:
            frame = getattr(app_instance, frame_name, None)
            if frame:
                frame.configure(fg_color=colors["grey_blue"])

    @stati

... (content truncated) ...
```

7. Tests

Test files and test configurations

■ *tests__init__.py*

Type: Python Module

■ *tests\conftest.py*

Type: Python Module

```
import os

import pytest
from openpyxl import Workbook


@pytest.fixture(scope="session", autouse=True)
def create_dummy_excel():
    filename = "./_Rx Repricing_wf.xlsx"
    if not os.path.exists(filename):
        wb = Workbook()
        ws = wb.active
        ws.title = "Claims Table"
        ws.append(
            ["pharmacy_npi", "pharmacy_nabp", "pharmacy_id"]
        )  # add columns as needed
        ws.append([1234567890, "NABP123", "123"])  # add dummy data as needed
        wb.save(filename)
    yield
    # Optionally, remove the file after tests
    # os.remove(filename)
```

■ *tests\pytest*

Type: File

Binary or special file - content not displayed

■ *tests\test_app.py*

Type: Python Module

```
import json
import os
import tkinter as tk

import pandas as pd
import pytest

from app import App, ConfigManager


@pytest.fixture
def tmp_work_dir(tmp_path, monkeypatch):
    """
    Create a temporary working directory and chdir into it, so that ConfigManager
    will create its config.json there.
    """
    monkeypatch.chdir(tmp_path)
    return tmp_path
```

```
def test_save_default_creates_config(tmp_work_dir):
    # When no config.json exists, ConfigManager.save_default() should create one
    # under the current directory (tmp_work_dir).
    cm = ConfigManager()
    config_path = tmp_work_dir / "config.json"

    assert config_path.exists(), "Config file was not created."
    with open(config_path, "r") as f:
        data = json.load(f)
    # By default, ConfigManager sets 'last_folder' to the cwd
    assert "last_folder" in data
    assert data["last_folder"] == str(tmp_work_dir)


def test_load_existing_config(tmp_work_dir):
    # Write a custom config.json, then ensure ConfigManager.load() honors it.
    config_path = tmp_work_dir / "config.json"
    custom = {"last_folder": "C:/some/path/xyz"}
    with open(config_path, "w") as f:
        json.dump(custom, f)

    cm = ConfigManager()
    assert (
        cm.config == custom
    ), "ConfigManager did not load the existing config.json correctly."


def test_filter_template_columns_extracts_correct_range():
    # Build a sample DataFrame where columns go: ['A','B','Client Name','X','Y','Logic','Z','W']
    df = pd.DataFrame(
        {
            "A": [1],
            "B": [2],
            "Client Name": ["foo"],
            "X": [3],
            "Y": [4],
            "Logic": [5],
            "Z": [6],
            "W": [7],
        }
    )

    # We only expect columns from 'Client Name' up through 'Logic' (inclusive).
    root = tk.Tk()
    root.withdraw()
    app = App(root)
    filtered = app.filter_template_columns(df)
    root.destroy()

    assert list(filtered.columns) == [
        "Client Name",
        "X",
        "Y",
        "Logic",
    ], f"Expected columns from 'Client Name' to 'Logic', got {list(filtered.columns)}"


def test_filter_template_columns_fallback_to_full_df_if_missing_logic():
    # If 'Client Name' or 'Logic' aren't found, it should return the full DataFrame unmodified
    df = pd.DataFrame({"Foo": [1, 2], "Bar": [3, 4]})

    root = tk.Tk()
    root.withdraw()
    app = App(root)
    result = app.filter_template_columns(df)
    root.destroy()

    # Since 'Client Name' or 'Logic' are not present, filter_template_columns should catch ValueError
    # and return the original DataFrame
    pd.testing.assert_frame_equal(result, df)


def test_format_dataframe_converts_datetimes_and_handles_na():
    # Build a DataFrame with one datetime column and one column containing a None
    orig = pd.DataFrame(
        {
            "dt1": [
```

```
                pd.to_datetime("2020-12-31 13:45:00"),
                pd.to_datetime("2021-01-01 00:00:00"),
            ],

... (content truncated) ...
```

■ tests\test_bg_disruption.py

Type: Python Module

```
from bg_disruption import process_data


def test_process_bg_runs():
    process_data()
```

■ tests\test_epls_lbl.py

Type: Python Module

```
from epls_lbl import main


def test_epls_main_runs():
    main()
```

■ tests\test_merge.py

Type: Python Module

```
from pathlib import Path

import pytest

from merge import merge_files


def test_merge_files(tmp_path):
    file1 = tmp_path / "f1.csv"
    file2 = tmp_path / "f2.csv"
    file1.write_text("DATEFILLED,SOURCERECORDID\n2020-01-01,1")
    file2.write_text("SOURCERECORDID,Total AWP (Historical)\n1,50.0")
    merge_files(str(file1), str(file2))
    assert Path("merged_file.xlsx").exists()
```

■ tests\test_openmdf_bg.py

Type: Python Module

```
from openmdf_bg import process_data


def test_process_openmdf_bg_runs():
    process_data()
```

■ tests\test_openmdf_tier.py

Type: Python Module

```
from openmdf_tier import process_data


def test_process_openmdf_tier_runs():
    process_data()
```

tests\test_sharx_lbl.py

Type: Python Module

```
from sharx_lbl import main


def test_sharx_main_runs():
    main()
```

tests\test_tier_disruption.py

Type: Python Module

```
from tier_disruption import process_data


def test_process_tier_runs():
    process_data()
```

tests\test_utils.py

Type: Python Module

tests\tier_disruption.py

Type: Python Module

```
import json
import os

import pandas as pd

# Standardize IDs


def standardize_pharmacy_ids(df):
    if "PHARMACYNPI" in df.columns:
        df["PHARMACYNPI"] = df["PHARMACYNPI"].astype(str).str.zfill(10)
    if "NABP" in df.columns:
        df["NABP"] = df["NABP"].astype(str).str.zfill(7)
    return df


def standardize_network_ids(network):
    if "pharmacy_npi" in network.columns:
        network["pharmacy_npi"] = network["pharmacy_npi"].astype(str).str.zfill(10)
    if "pharmacy_nabp" in network.columns:
        network["pharmacy_nabp"] = network["pharmacy_nabp"].astype(str).str.zfill(7)
    return network


def merge_with_network(df, network):
    return df.merge(
        network,
```

```
        left_on=["PHARMACYNPI", "NABP"],
        right_on=["pharmacy_npi", "pharmacy_nabp"],
        how="left",
    )


def load_file_paths(json_file):
    with open(json_file, "r") as f:
        file_paths = json.load(f)
    return file_paths


def summarize_by_tier(df, col, from_val, to_val):
    filtered = df[(df[col] == from_val) & (df["FormularyTier"] == to_val)]
    pt = pd.pivot_table(
        filtered,
        values=["Rxs", "MemberID"],
        index=["Product Name"],
        aggfunc={"Rxs": "sum", "MemberID": pd.Series.nunique},
    )
    rxs = filtered["Rxs"].sum()
    members = filtered["MemberID"].nunique()
    return pt, rxs, members


def process_data():
    file_paths = load_file_paths("file_paths.json")

    # Only load claims if the path is present and not empty
    claims = None
    if "reprice" in file_paths and file_paths["reprice"]:
        claims = pd.read_excel(file_paths["reprice"], sheet_name="Claims Table")
    else:
        print("No reprice/template file provided. Skipping claims loading.")
        # You can decide to return, raise, or continue with alternate logic here
        return

    medi = pd.read_excel(file_paths["medi_span"])[
        ["NDC", "Maint Drug?", "Product Name"]
    ]
    u = pd.read_excel(file_paths["u_disrupt"], sheet_name="Universal NDC")[
        ["NDC", "Tier"]
    ]
    e = pd.read_excel(file_paths["e_disrupt"], sheet_name="Alternatives NDC")[
        ["NDC", "Tier", "Alternative"]
    ]
    network = pd.read_excel(file_paths["n_disrupt"])[
        ["pharmacy_npi", "pharmacy_nabp", "pharmacy_is_excluded"]
    ]

    df = claims.merge(medi, on="NDC", how="left")
    df = df.merge(u.rename(columns={"Tier": "Universal Tier"}), on="NDC", how="left")
    df = df.merge(e.rename(columns={"Tier": "Exclusive Tier"}), on="NDC", how="left")

    df = standardize_pharmacy_ids(df)
    network = standardize_network_ids(network)
    df = merge_with_network(df, network)

    df["DATEFILLED"] = pd.to_datetime(df["DATEFILLED"], errors="coerce")
    df = df.drop_duplicates()
    df["Logic"] = pd.to_numeric(df["Logic"], errors="coerce")
    df["FormularyTier"] = pd.to_numeric(df["FormularyTier"], errors="coerce")

    latest_date = df["DATEFILLED"].max()
    starting_point = latest_date - pd.DateOffset(months=6) + pd.DateOffset(

... (content truncated) ...
```

■ *tests\unittest.mock*

Type: .MOCK File

Binary or special file - content not displayed

8. Resources

Documentation, guides, and resource files

■ *resources\AUDIT_SYSTEM_GUIDE.md*

Type: Documentation

Enhanced Audit and Error Logging System

Overview

Your Repricing Automation Program now has a comprehensive audit and error logging system that will help you assist users effectively.

■ ***What the System Logs***

1. **User Identification & Session Tracking*

- **Username**: Captured via `getpass.getuser()` (e.g., "DamionMorrison")
- **Computer Name**: Host machine identifier (e.g., "L01403-DATA")
- **Session Start/End**: Complete session lifecycle tracking
- **System Information**: Python version, OS details for troubleshooting

2. **File Operations*

- **File Imports**: Tracks when users import File1, File2, and templates
- **File Paths**: Records exact file locations accessed
- **File Errors**: Logs permission issues, missing files, corrupt data

3. **Process Tracking*

- **Process Start/Stop**: When repricing processes begin and end
- **Process Errors**: Failures during data processing, merging, or calculations
- **User Actions**: What the user was trying to do when errors occurred

4. **Error Categories for Support*

- **USER_ERROR**: User-related issues (wrong files, invalid inputs)
- **SYSTEM_ERROR**: Application/system failures (memory, crashes)
- **FILE_ERROR**: File access, permission, or format issues
- **DATA_ERROR**: Data processing, validation, or calculation errors

■ ***Support Tools Available***

1. **Error Analysis Tool** (`safe_error_analysis.py`)

```bash
python safe_error_analysis.py
```

Provides:
- Summary of all errors in the last 7 days
- Breakdown by error type and affected users
- Most problematic scripts/functions
- Recent successful operations

2. **User Support Report** (`user_support_report.py`)

```bash
python user_support_report.py
```

Or modify to check specific user:

python -c "from user_support_report import generate_user_support_report; generate_user_support_report('Username')"
```
```

Provides:
- User-specific error history
- Detailed error context and system information
- Timeline of issues for pattern identification

3. **Real-time Audit Log Viewer** (In Application)

- Click "Shared Audit Log" button in the application
- Live updating view of all user activities
- Search functionality to filter specific users or error types

■ ***Example Error Log Entry***

```csv
Timestamp,User,Script,Message,Status
2025-07-10 11:18:32,DamionMorrison,File1Import,"USER ERROR - FILE_NOT_FOUND | User: DamionMorrison on L01403-DATA | Python: 3.12.10 | OS: Windows 11 | Action: Import File | File: missing_data.xlsx | Error: Could not find file",USER_ERROR
```

■ ***How to Help Users***

1. **When a User Reports an Issue:*

1. Ask for their **username** and **approximate time** of the error
2. Run the support report: `generate_user_support_report("Username")`
3. Check the audit log for their recent activities

2. **Common Issue Patterns:*

File Errors*

- **Symptoms**: "Cannot import file" or "File not found"
- **Check**: File paths, permissions, OneDrive sync status
- **Log Shows**: Exact f

... (content truncated) ...

■ *resources\AUDIT_SYSTEM_GUIDE.pdf*

Type: .PDF File

Binary or special file - content not displayed

■ resources\CLAIM_DETAIL_LOGIC_INTEGRATION.md

Type: Documentation

■ resources\README.pdf

Type: .PDF File

Binary or special file - content not displayed

■ resources\README.txt

Type: Documentation

Repricing Automation Toolkit (Enhanced)

This package contains the fully optimized and enhanced version of your Python-based Repricing Automation system, with improvements in structure, logging, performance profiling, and configuration management.

■ *Contents*

File	Purpose
`app.py`	Main GUI entry point with Excel templating, async paste, audit logging, dark mode, and real-time progress
`merge.py`	Merges two datasets, applies formatting
`openmdf_tier.py`	Processes Open MDF Tier logic
`openmdf_bg.py`	Processes Open MDF Brand/Generic logic
`tier_disruption.py`	Standard tier disruption processor
`bg_disruption.py`	Brand/Generic disruption processor
`sharx_lbl.py`	Generates SHARx Line By Line output
`epls_lbl.py`	Generates EPLS Line By Line output
`mp_helpers.py`	Multiprocessing helper for identifying reversals and matching claims with 'OR' logic
`utils.py`	Common utilities (logging, ID standardization, filtering)
`config_loader.py`	Centralized config resolution via OneDrive
`audit_helper.py`	Safe wrapper around audit logging (`make_audit_entry`)
`excel_utils.py`	Async/safe Excel handling via xlwings with COM fallback
`profile_runner.py`	Runs performance profiling on `app.py` using `cProfile`

■ *Setup Instructions*

1. Environment Requirements

- Python 3.13.5 (64-bit) or newer
- Dependencies:
- `pandas`, `openpyxl`, `xlsxwriter`, `customtkinter`, `plyer`, `pywin32`, `numba`, `pyarrow`, `xlwings`

Use pip to install all dependencies:
```bash
```

pip install pandas openpyxl xlsxwriter plyer numba pywin32 customtkinter xlwings pyarrow
```

> Windows-only features like `win32com.client` require `pywin32`.
> Async Excel support via `xlwings` will fallback to COM automation if necessary.

---

## ■ *Usage Guide*

### *Launch the Application*

```bash
python app.py
```

Run a Performance Profile

```bash
python profile_runner.py
```

Generates `profile_stats.prof` and prints slowest calls.

■ *Audit Logging*

All scripts log user actions/errors to:

```
%OneDrive%/True Community - Data Analyst/Python Repricing Automation Program/Logs/audit_log.csv
```

If unreachable, logs fall back to:

```
local_fallback_log.txt
```

■ *Configuration Management*

To manage file paths centrally, update:

```json
file_paths.json
```

Or adapt `config_loader.py` to replace JSON with `.env` or `.toml` later if preferred.

■ *Additional Notes*

- Logs are rotated (`utils.log`) to avoid file bloat.
- All enhancements maintain full backward compatibility with your original logic.
- UI now supports full dark/light mode toggle and async Excel pasting with live progress.
- For setup, troubleshooting, and advanced configuration, see `SETUP_GUIDE.txt` and the walkthrough documentation.
```

For questions or future updates, contact you

... (content truncated) ...

■ resources\SETUP_GUIDE.pdf

Type: .PDF File

Binary or special file - content not displayed

■ resources\SETUP_GUIDE.txt

Type: Documentation

Repricing Automation Program – Setup Guide (Internal Users)

This guide helps internal users install, configure, and run the Repricing Automation Program on a Windows-based system. All steps are streamlined for non-developers.

■ *1. System Requirements*

Requirement	Description
OS	Windows 10 or later
Python	Version 3.13.5 (64-bit)
Excel	Microsoft Excel (macros enabled)
OneDrive	Sync enabled and signed in

■ *2. Software Installation*

A. Install Python (if not installed)

1. Visit: https://www.python.org/downloads/windows/
2. Download **Python 3.13.5 (64-bit)**.
3. During installation, check **“Add Python to PATH”**.
4. Complete the installation.

B. Install Required Python Packages

1. Open **Command Prompt** as Administrator.
2. Run the following command (copy and paste as one line):

```bash
pip install pandas openpyxl xlsxwriter plyer numba pywin32 customtkinter xlwings pyarrow
```

> If you see errors, ensure you are using the correct version of Python and have internet access.

■ *3. Folder Setup*

- Place all program files in a single folder (e.g., `C:\Users\\RepricingAutomation`).
- Ensure you have write access to this folder.

- Make sure your OneDrive is running and synced.

■ *4. Configuration*

- **File Paths:**
Edit `file_paths.json` if you need to customize where input/output files are stored.
- **Excel Templates:**
Keep the template file named `_Rx Repricing_wf.xlsx` unless instructed otherwise.

■ *5. Running the Program*

1. Double-click or run `app.py`:
- Open Command Prompt in the program folder.
- Run:
```bash
python app.py
```
2. Use the GUI to import files, select disruption type, and start processing.

■ *6. Troubleshooting*

- **Excel Errors:**
- Close all open Excel windows before running the program.
- Ensure macros are enabled in Excel.
- **Python Not Found:**
- Reboot your computer after installing Python.
- Make sure Python is added to your PATH.
- **OneDrive Issues:**
- Confirm you are signed in and syncing.
- Check that logs are being written to the correct OneDrive folder.

■ *7. Support*

- For help, contact your IT department or the program maintainer.
- Provide screenshots and any error messages for faster support.

If you need more advanced configuration or want to run performance profiling, see the main walkthrough documentation.

■ *resources\UW_Automation_Program_Documentation.pdf*

Type: .PDF File

Binary or special file - content not displayed

■ *resources\generate_audit_guide_pdf.py*

Type: Python Module

```
"""
Generate PDF from AUDIT_SYSTEM_GUIDE.md
"""

import markdown
from weasyprint import HTML
from pathlib import Path

def generate_audit_guide_pdf():
    """Generate PDF from the AUDIT_SYSTEM_GUIDE.md file."""

    # Get the current directory
    current_dir = Path(__file__).parent

    # Input and output file paths
    md_file = current_dir / "AUDIT_SYSTEM_GUIDE.md"
    pdf_file = current_dir / "AUDIT_SYSTEM_GUIDE.pdf"

    try:
        # Read the markdown file
        with open(md_file, 'r', encoding='utf-8') as f:
            md_content = f.read()

        # Convert markdown to HTML
        html_content = markdown.markdown(
            md_content,
            extensions=['codehilite', 'fenced_code', 'tables', 'toc']
        )

        # Create a complete HTML document with styling
        full_html = f"""
        <!DOCTYPE html>
        <html>
        <head>
            <meta charset="utf-8">
            <title>Enhanced Audit and Error Logging System Guide</title>
            <style>
                body {{
                    font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
                    line-height: 1.6;
                    margin: 40px;
                    color: #333;
                    max-width: 800px;
                }}

                h1 {{
                    color: #2c3e50;
                    border-bottom: 3px solid #3498db;
                    padding-bottom: 10px;
                    margin-top: 30px;
                }}

                h2 {{
                    color: #34495e;
                    border-bottom: 2px solid #ecf0f1;
                    padding-bottom: 5px;
                    margin-top: 25px;
                }}

                h3 {{
                    color: #2980b9;
                    margin-top: 20px;
                }}

                h4 {{
                    color: #27ae60;
                    margin-top: 15px;
                }}

                code {{
                    background-color: #f8f9fa;
                    padding: 2px 4px;
                    border-radius: 3px;
                    font-family: 'Consolas', 'Monaco', monospace;
                    color: #d63384;
                }}
```

```
                pre {{
                    background-color: #f8f9fa;
                    border: 1px solid #e9ecef;
                    border-radius: 5px;
                    padding: 15px;
                    overflow-x: auto;
                    margin: 15px 0;
                }}

                pre code {{
                    background-color: transparent;
                    color: #212529;
                    padding: 0;
                }}

                ul, ol {{
                    margin: 10px 0;
                    padding-left: 25px;
                }}

                li {{
                    margin: 5px

... (content truncated) ...
```

■ resources\generate_audit_guide_pdf_simple.py

Type: Python Module

```
"""
Generate PDF from AUDIT_SYSTEM_GUIDE.md using reportlab
"""

import re
from pathlib import Path
from reportlab.lib.pagesizes import A4
from reportlab.platypus import SimpleDocTemplate, Paragraph, Spacer, Preformatted
from reportlab.lib.styles import getSampleStyleSheet, ParagraphStyle
from reportlab.lib.colors import HexColor

def generate_audit_guide_pdf():
    """Generate PDF from the AUDIT_SYSTEM_GUIDE.md file using ReportLab."""

    # Get the current directory
    current_dir = Path(__file__).parent

    # Input and output file paths
    md_file = current_dir / "AUDIT_SYSTEM_GUIDE.md"
    pdf_file = current_dir / "AUDIT_SYSTEM_GUIDE.pdf"

    try:
        # Read the markdown file
        with open(md_file, 'r', encoding='utf-8') as f:
            content = f.read()

        # Create PDF document
        doc = SimpleDocTemplate(
            str(pdf_file),
            pagesize=A4,
            rightMargin=72,
            leftMargin=72,
            topMargin=72,
            bottomMargin=18
        )

        # Get styles
        styles = getSampleStyleSheet()

        # Create custom styles
        title_style = ParagraphStyle(
            'CustomTitle',
            parent=styles['Title'],
            fontSize=20,
            spaceAfter=20,
```

```
            textColor=HexColor('#2c3e50'),
            alignment=1  # Center
        )

        heading1_style = ParagraphStyle(
            'CustomHeading1',
            parent=styles['Heading1'],
            fontSize=16,
            spaceAfter=12,
            spaceBefore=20,
            textColor=HexColor('#2c3e50')
        )

        heading2_style = ParagraphStyle(
            'CustomHeading2',
            parent=styles['Heading2'],
            fontSize=14,
            spaceAfter=10,
            spaceBefore=15,
            textColor=HexColor('#34495e')
        )

        heading3_style = ParagraphStyle(
            'CustomHeading3',
            parent=styles['Heading3'],
            fontSize=12,
            spaceAfter=8,
            spaceBefore=12,
            textColor=HexColor('#2980b9')
        )

        code_style = ParagraphStyle(
            'Code',
            parent=styles['Code'],
            fontSize=9,
            fontName='Courier',
            backColor=HexColor('#f8f9fa'),
            borderColor=HexColor('#e9ecef'),
            borderWidth=1,
            borderPadding=10,
            leftIndent=10,
            rightIndent=10,
            spaceAfter=10
        )

        # Story to hold document content
        story = []

        # Split content into lines
        lines = content.split('\n')

        i = 0
        while i < len(lines):
            line = lines[i].strip()

            if not line:
                story.append(Spacer(1, 6))
                i += 1
                continue

            # Handle headings
            if line.startswith('# '):
                if

... (content truncated) ...
```

■ *resources\generate_project_documentation.py*

Type: Python Module

```
"""
Generate comprehensive PDF documentation for the entire UW Automation Program directory
"""
```

```
import os
from pathlib import Path
from reportlab.lib.pagesizes import A4
from reportlab.platypus import SimpleDocTemplate, Paragraph, Spacer, Preformatted, PageBreak, Table, Table
from reportlab.lib.styles import getSampleStyleSheet, ParagraphStyle
from reportlab.lib.colors import HexColor, black, white
from reportlab.lib.units import inch
from datetime import datetime

def generate_project_documentation_pdf():
    """Generate comprehensive PDF documentation for the entire project."""

    # Get the project root directory (one level up from resources)
    current_dir = Path(__file__).parent
    project_root = current_dir.parent

    # Output file
    pdf_file = current_dir / "UW_Automation_Program_Documentation.pdf"

    try:
        # Create PDF document
        doc = SimpleDocTemplate(
            str(pdf_file),
            pagesize=A4,
            rightMargin=72,
            leftMargin=72,
            topMargin=72,
            bottomMargin=72
        )

        # Get styles
        styles = getSampleStyleSheet()

        # Create custom styles
        title_style = ParagraphStyle(
            'ProjectTitle',
            parent=styles['Title'],
            fontSize=24,
            spaceAfter=30,
            textColor=HexColor('#2c3e50'),
            alignment=1  # Center
        )

        subtitle_style = ParagraphStyle(
            'Subtitle',
            parent=styles['Heading1'],
            fontSize=14,
            spaceAfter=20,
            textColor=HexColor('#7f8c8d'),
            alignment=1  # Center
        )

        section_style = ParagraphStyle(
            'SectionHeading',
            parent=styles['Heading1'],
            fontSize=18,
            spaceAfter=15,
            spaceBefore=25,
            textColor=HexColor('#2c3e50'),
            borderWidth=2,
            borderColor=HexColor('#3498db'),
            borderPadding=5
        )

        # subsection_style is not used and has been removed

        file_header_style = ParagraphStyle(
            'FileHeader',
            parent=styles['Heading3'],
            fontSize=12,
            spaceAfter=8,
            spaceBefore=12,
            textColor=HexColor('#2980b9'),
            backColor=HexColor('#ecf0f1'),
            borderWidth=1,
            borderColor=HexColor('#bdc3c7'),
            borderPadding=5
```

```
        )

        code_style = ParagraphStyle(
            'CodeBlock',
            parent=styles['Code'],
            fontSize=8,
            fontName='Courier',
            backColor=HexColor('#f8f9fa'),
            borderColor=HexColor('#e9ecef'),
            borderWidth=1,
            borderPadding=10,
            leftIndent=10,
            rightIndent=10,
            spaceAfter=10
        )

        # Story to hold document content
        story = []

        # Title page


... (content truncated) ...
```

■ resources\generate_readme_pdf.py

Type: Python Module

```
from fpdf import FPDF
import re

with open("README.txt", "r", encoding="utf-8") as f:
    lines = f.readlines()


def clean_text(text):
    # Replace en dash and other unicode dashes with hyphen
    text = re.sub(r"[\u2013\u2014\u2012]", "-", text)
    # Replace curly quotes with straight quotes
    text = text.replace("“", '"').replace("”", '"').replace("‘", "'").replace("’", "'")
    # Remove emoji and non-ascii characters
    text = re.sub(r"[^\x00-\x7F]+", "", text)
    return text


pdf = FPDF()
pdf.set_auto_page_break(auto=True, margin=15)
pdf.add_page()
pdf.set_font("Arial", size=12)

for line in lines:
    line = clean_text(line)
    if line.startswith("# "):
        pdf.set_font("Arial", "B", 16)
        pdf.cell(0, 10, line[2:].strip(), ln=True)
        pdf.set_font("Arial", size=12)
    elif line.startswith("## "):
        pdf.set_font("Arial", "B", 14)
        pdf.cell(0, 8, line[3:].strip(), ln=True)
        pdf.set_font("Arial", size=12)
    elif line.startswith("### "):
        pdf.set_font("Arial", "B", 12)
        pdf.cell(0, 8, line[4:].strip(), ln=True)
        pdf.set_font("Arial", size=12)
    elif line.startswith("- ") or line.startswith("1.") or line.startswith("    "):
        pdf.multi_cell(0, 8, line.strip())
    elif line.strip() == "":
        pdf.ln(2)
    else:
        pdf.multi_cell(0, 8, line.strip())

pdf.output("README.pdf")
print("PDF created: README.pdf")
```

■ resources\generate_setup_pdf.py

Type: Python Module

```
from fpdf import FPDF
import re

with open("SETUP_GUIDE.txt", "r", encoding="utf-8") as f:
    lines = f.readlines()


def clean_text(text):
    # Replace en dash and other unicode dashes with hyphen
    text = re.sub(r"[\u2013\u2014\u2012]", "-", text)
    # Replace curly quotes with straight quotes
    text = text.replace("“", '"').replace("”", '"').replace("‘", "'").replace("’", "'")
    # Remove emoji and non-ascii characters
    text = re.sub(r"[^\x00-\x7F]+", "", text)
    return text


pdf = FPDF()
pdf.set_auto_page_break(auto=True, margin=15)
pdf.add_page()
pdf.set_font("Arial", size=12)

for line in lines:
    line = clean_text(line)
    if line.startswith("# "):
        pdf.set_font("Arial", "B", 16)
        pdf.cell(0, 10, line[2:].strip(), ln=True)
        pdf.set_font("Arial", size=12)
    elif line.startswith("## "):
        pdf.set_font("Arial", "B", 14)
        pdf.cell(0, 8, line[3:].strip(), ln=True)
        pdf.set_font("Arial", size=12)
    elif line.startswith("### "):
        pdf.set_font("Arial", "B", 12)
        pdf.cell(0, 8, line[4:].strip(), ln=True)
        pdf.set_font("Arial", size=12)
    elif line.startswith("- ") or line.startswith("1.") or line.startswith("    "):
        pdf.multi_cell(0, 8, line.strip())
    elif line.strip() == "":
        pdf.ln(2)
    else:
        pdf.multi_cell(0, 8, line.strip())

pdf.output("SETUP_GUIDE.pdf")
print("PDF created: SETUP_GUIDE.pdf")
```

■ resources\generate_walkthrough_pdf.py

Type: Python Module

```
from fpdf import FPDF
import re

# Read the markdown file
with open("repricing_walkthrough.md", "r", encoding="utf-8") as f:
    lines = f.readlines()


def clean_text(text):
    # Replace en dash and other unicode dashes with hyphen
    text = re.sub(r"[\u2013\u2014\u2012]", "-", text)
    # Replace curly quotes with straight quotes
    text = text.replace("“", '"').replace("”", '"').replace("‘", "'").replace("’", "'")
    return text

pdf = FPDF()
```

```
pdf.set_auto_page_break(auto=True, margin=15)
pdf.add_page()
pdf.set_font("Arial", size=12)

for line in lines:
    line = clean_text(line)
    # Add section headers in bold
    if line.startswith("# "):
        pdf.set_font("Arial", "B", 16)
        pdf.cell(0, 10, line[2:].strip(), ln=True)
        pdf.set_font("Arial", size=12)
    elif line.startswith("## "):
        pdf.set_font("Arial", "B", 14)
        pdf.cell(0, 8, line[3:].strip(), ln=True)
        pdf.set_font("Arial", size=12)
    elif line.startswith("### "):
        pdf.set_font("Arial", "B", 12)
        pdf.cell(0, 8, line[4:].strip(), ln=True)
        pdf.set_font("Arial", size=12)
    elif line.startswith("- ") or line.startswith("1.") or line.startswith("    "):
        pdf.multi_cell(0, 8, line.strip())
    elif line.strip() == "":
        pdf.ln(2)
    else:
        pdf.multi_cell(0, 8, line.strip())

pdf.output("repricing_walkthrough.pdf")
print("PDF created: repricing_walkthrough.pdf")
```

■ *resources\profile_stats.prof*

Type: .PROF File

Binary or special file - content not displayed

... and 3 more files in this category

9. Complete File Structure

Complete directory tree of the project:

```
■■■ config
■   ■■■ __init__.py
■   ■■■ app_config.py
■   ■■■ config.json
■   ■■■ config_loader.py
■   ■■■ file_paths.json
■   ■■■ improved_config_manager.py
■■■ docs
■■■ modules
■   ■■■ __init__.py
■   ■■■ audit_helper.py
■   ■■■ bg_disruption.py
■   ■■■ check_audit.py
■   ■■■ data_processor.py
■   ■■■ epls_lbl.py
■   ■■■ error_analysis_tool.py
■   ■■■ file_processor.py
■   ■■■ log_manager.py
■   ■■■ merge.py
■   ■■■ merge_log.log
■   ■■■ merged_file.xlsx
■   ■■■ mp_helpers.py
■   ■■■ openmdf_bg.py
■   ■■■ openmdf_bg_simple.py
■   ■■■ openmdf_tier.py
■   ■■■ process_manager.py
■   ■■■ profile_runner.py
■   ■■■ safe_error_analysis.py
■   ■■■ sharx_lbl.py
■   ■■■ template_processor.py
■   ■■■ tier_disruption.py
■   ■■■ ui_builder.py
■   ■■■ user_support_report.py
■■■ resources
■   ■■■ AUDIT_SYSTEM_GUIDE.md
■   ■■■ AUDIT_SYSTEM_GUIDE.pdf
■   ■■■ CLAIM_DETAIL_LOGIC_INTEGRATION.md
■   ■■■ generate_audit_guide_pdf.py
■   ■■■ generate_audit_guide_pdf_simple.py
■   ■■■ generate_project_documentation.py
■   ■■■ generate_readme_pdf.py
■   ■■■ generate_setup_pdf.py
■   ■■■ generate_walkthrough_pdf.py
■   ■■■ profile_stats.prof
■   ■■■ README.pdf
■   ■■■ README.txt
■   ■■■ repricing_walkthrough.md
■   ■■■ repricing_walkthrough.pdf
■   ■■■ requirements.txt
■   ■■■ SETUP_GUIDE.pdf
■   ■■■ SETUP_GUIDE.txt
■   ■■■ UW_Automation_Program_Documentation.pdf
■■■ tests
■   ■■■ __init__.py
■   ■■■ conftest.py
■   ■■■ pytest
■   ■■■ test_app.py
■   ■■■ test_bg_disruption.py
■   ■■■ test_epls_lbl.py
■   ■■■ test_merge.py
■   ■■■ test_openmdf_bg.py
■   ■■■ test_openmdf_tier.py
■   ■■■ test_sharx_lbl.py
■   ■■■ test_tier_disruption.py
■   ■■■ test_utils.py
■   ■■■ tier_disruption.py
■   ■■■ unittest.mock
■■■ ui
■   ■■■ ui_components.py
■■■ utils
■   ■■■ __init__.py
```

```
■   ■■■ excel_utils.py
■   ■■■ logic_processor.py
■   ■■■ utils.py
■   ■■■ utils_functions.py
■■■ app.py
■■■ config.json
■■■ mover.py
```