UW Automation Program

Repricing Walkthrough Guide

Generated on: July 11, 2025 at 11:39 AM

Repricing Automation Toolkit - Comprehensive Walkthrough

Overview

The Repricing Automation Toolkit is designed for Windows, with a graphical user interface (GUI) for automating complex Excel-based repricing workflows. It is tailored for pharmacy/claims data, supporting disruption analysis, audit logging, and advanced Excel handling.

Main Components & Files

- app.py: Main GUI entry point with Excel templating, async paste, audit logging, dark mode, and real-time progress
- merge.py: Merges two datasets and applies formatting
- openmdftier.py / openmdfbg.py: Process Open MDF Tier and Brand/Generic logic, respectively
- tierdisruption.py / bgdisruption.py: Handle standard tier and brand/generic disruption processing
- sharxlbl.py / eplslbl.py: Generate SHARx and EPLS Line By Line outputs
- mphelpers.py: Multiprocessing helpers for matching claims and reversals
- excelutils.py: Async and safe Excel handling, using xlwings with a COM fallback
- utils.py: Common utilities (logging, filtering, ID standardization)
- configloader.py: Centralized config management (file paths, etc.)
- audithelper.py: Safe wrapper for audit logging
- profilerunner.py: Runs performance profiling on the main app

User Interface (GUI)

- File Import Buttons: Import two main data files and a template file
- Disruption Type Buttons: Choose between Tier Disruption, B/G Disruption, OpenMDF (Tier), and OpenMDF (B/G)
- Process Buttons:
- Start Repricing: Begins the main repricing workflow
- Generate SHARx/EPLS LBL: Produces line-by-line outputs for SHARx/EPLS
- Cancel: Stops a running process
- View Logs / Shared Audit Log: Opens log viewers
- Switch to Dark Mode: Toggles UI theme
- Exit: Closes the application
- Progress Bar & Status: Shows real-time progress and messages
- Notes Section: Displays important reminders (e.g., close Excel, keep template name)

Core Capabilities

- Automated Excel Processing: Reads, writes, and formats Excel files, including async pasting and live progress
- Disruption Analysis: Runs various disruption analyses (tier, brand/generic, OpenMDF) with a single click
- Audit Logging: All actions and errors are logged to a central CSV (with OneDrive fallback)
- Multiprocessing: Uses multiprocessing for heavy data operations (e.g., matching claims)
- Configurable Paths: All file paths are managed via filepaths.json and configloader.py
- Performance Profiling: Run profilerunner.py to analyze and optimize performance
- Dark/Light Mode: Full UI theming support
- Error Handling: Robust error handling and user notifications (including desktop notifications via plyer)

Setup & Requirements

- OS: Windows 10 or later
- Python: 3.13.5 (or 3.10+)
- Excel: With macros enabled
- OneDrive: For log syncing

Install dependencies:

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

How to Use

1. Launch the App:

```
python app.py
```

2. Import Files: Use the GUI to import your data and template files
3. Select Disruption Type: Click the relevant button for your analysis
4. Start Repricing: Click "Start Repricing" to run the workflow
5. Generate LBLs: Use SHARx/EPLS buttons for line-by-line outputs
6. View Logs: Access logs for audit and troubleshooting
7. Switch Theme: Toggle between dark and light mode as needed

Logging & Audit

All actions/errors are logged to:
%OneDrive%/True Community - Data Analyst/Python Repricing Automation Program/Logs/auditlog.csv
If OneDrive is unavailable, logs fall back to localfallbacklog.txt.

Configuration

- File Paths: Managed in filepaths.json

- Config Loader: configloader.py can be adapted for other config formats

Advanced

- Performance Profiling:

python profilerunner.py
Generates profilestats.prof for performance analysis
- Excel Handling: Uses xlwings if available, otherwise falls back to COM automation

Additional Notes

- Logs are rotated to prevent bloat
- All enhancements are backward compatible
- UI is modern, with async Excel pasting and live progress

In-Depth Breakdown of Each Python File

app.py

Purpose:
The main entry point and GUI for the toolkit. Handles user interaction, file imports, process control, and orchestrates all major workflows.

Key Functions & Classes:
- App class: Central GUI logic, event handlers, and workflow management.
- buildui(): Constructs the GUI (buttons, progress bar, notes, etc.).
- importfile1, importfile2, importtemplatefile: File import dialogs.
- startprocessthreaded, startprocess, startprocessinternal: Launch and manage the main repricing process, including threading for responsiveness.
- startdisruption: Triggers disruption analysis based on user selection.
- sharxlbl, eplslbl: Generate SHARx/EPLS line-by-line outputs.
- showlogviewer, showsharedlogviewer: Display logs to the user.
- toggledarkmode: Switches between light and dark UI themes.
- cancelprocess: Allows the user to cancel a running process.
- writeauditlog: Records actions/errors for audit purposes.
- updateprogress: Updates the progress bar and status messages.
- ConfigManager class: Handles loading and managing configuration files.
- Logging setup: All actions/errors are logged for traceability.

Process:
The user interacts with the GUI to import files, select disruption types, and start processes. The app coordinates calls to other modules for data processing, Excel manipulation, and logging.

---

merge.py

Purpose:
Handles merging of two datasets and applies necessary formatting for downstream processing.

Key Functions:

- Likely contains a main function or class to:
- Read two input files (from the GUI).
- Merge them based on business logic (e.g., matching keys, deduplication).
- Clean and format the merged data for use in repricing or disruption analysis.
- Save the merged output for further processing.

Process:
Called by app.py when the user initiates a merge or repricing workflow.

---

openmdf_tier.py / openmdf_bg.py

Purpose:
Specialized processors for Open MDF logic:

- openmdftier.py: Handles tier-based Open MDF logic.
- openmdfbg.py: Handles brand/generic-based Open MDF logic.

Key Functions:

- Each file likely contains:
- Functions to load input data.
- Apply Open MDF-specific business rules (tier or brand/generic).
- Output processed data for further use or reporting.

Process:
Triggered by the corresponding disruption buttons in the GUI.

---

tier_disruption.py / bg_disruption.py

Purpose:
Standard disruption processors:

- tierdisruption.py: For tier-based disruption analysis.
- bgdisruption.py: For brand/generic disruption analysis.

Key Functions:

- Functions to:
- Load and validate input data.
- Apply disruption logic (e.g., compare old vs. new tiers, flag changes).
- Output results for reporting or further processing.

Process:
Called by the GUI when the user selects a disruption type and starts the process.

---

sharx_lbl.py / epls_lbl.py

Purpose:
Generate line-by-line (LBL) outputs for SHARx and EPLS, respectively.

Key Functions:

- Functions to:
- Read processed data.
- Format and output LBL files according to SHARx/EPLS requirements.
- Handle any special formatting or calculations needed for LBLs.

Process:
Triggered by the “Generate SHARx LBL” or “Generate EPLS LBL” buttons in the GUI.

---

mp_helpers.py

Purpose:
Multiprocessing helpers for performance-critical operations.

Key Functions:

- Functions to:
- Identify reversals and match claims using parallel processing.
- Implement “OR” logic for complex matching scenarios.
- Optimize heavy data operations to improve speed.

Process:
Used internally by disruption and merge modules to accelerate processing.

---

excel_utils.py

Purpose:
Safe and asynchronous Excel file handling.

Key Functions:

- Functions to:
- Open, read, and write Excel files using xlwings (with COM fallback).
- Handle Excel-specific formatting, cell coloring, and template management.
- Ensure safe file access (avoid file locks, handle open instances).

Process:
Called by all modules that need to interact with Excel files.

---

utils.py

Purpose:
General utility functions used across the project.

Key Functions:

- Logging helpers.

- ID standardization and filtering.
- Miscellaneous helpers for data cleaning, error handling, etc.

Process:
Imported and used by most other modules.

---

config_loader.py

Purpose:
Centralized configuration management.

Key Functions:
- Load file paths and settings from filepaths.json or other config files.
- Provide a single source of truth for paths and environment settings.

Process:
Used by app.py and other modules to resolve file locations and settings.

---

audit_helper.py

Purpose:
Safe wrapper for audit logging.

Key Functions:
- makeauditentry: Write audit entries to the log file, ensuring no data loss or corruption.
- Handle log file rotation and fallback if the main log is unavailable.

Process:
Called by app.py and other modules whenever an action or error needs to be logged.

---

profile_runner.py

Purpose:
Performance profiling for the main application.

Key Functions:
- Runs cProfile or similar profiling tools on app.py.
- Outputs performance stats to profilestats.prof and prints slowest calls.

Process:
Run manually to analyze and optimize performance bottlenecks.

Function-by-Function Breakdown

app.py

- ConfigManager.init(self): Initializes the configuration manager, loads config file paths.

- ConfigManager.savedefault(self): Saves default configuration settings.
- ConfigManager.load(self): Loads configuration from file.
- ConfigManager.save(self): Saves current configuration to file.
- App.init(self, root): Initializes the main GUI, sets up variables, and builds the UI.
- App.applythemecolors(self, colors): Applies the selected color theme to the UI.
- App.checktemplate(self, filepath): Validates the template file for required structure.
- App.sharxlbl(self): Generates SHARx line-by-line output from processed data.
- App.eplslbl(self): Generates EPLS line-by-line output from processed data.
- App.showsharedlogviewer(self): Displays the shared audit log to the user.
- App.buildui(self): Constructs all GUI elements (buttons, frames, labels, etc.).
- App.importfile1(self): Handles importing the first data file.
- App.importfile2(self): Handles importing the second data file.
- App.importtemplatefile(self): Handles importing the Excel template file.
- App.cancelprocess(self): Allows the user to cancel a running process.
- App.showlogviewerold(self): Displays the old version of the log viewer.
- App.toggledarkmode(self): Switches the UI between dark and light mode.
- App.updateprogress(self, value=None, message=None): Updates the progress bar and status label.

merge.py

- mergefiles(file1path, file2path): Merges two input files, applies business logic, and outputs a formatted merged file.

openmdf_tier.py

- processdata(): Processes input data using Open MDF Tier logic and outputs results.

openmdf_bg.py

- processdata(): Processes input data using Open MDF Brand/Generic logic and outputs results.
- pivotandcount(data): Helper function to pivot data and count relevant metrics.

tier_disruption.py

- summarizebytier(df, col, fromval, toval): Summarizes changes by tier between two values.
- processdata(): Runs the main tier disruption analysis and outputs results.

bg_disruption.py

- processdata(): Runs the main brand/generic disruption analysis and outputs results.
- pivot(d, includealternative=False): Pivots data for summary, optionally including alternatives.
- count(d): Counts unique members or relevant metrics in the data.

sharx_lbl.py

• showmessage(awp: float, ing: float, total: float, rxs: int): Displays a summary message for SHARx LBL output.

• main(): Main entry point for generating SHARx LBL output.

epls_lbl.py

• showmessage(awp, ing, total, rxs): Displays a summary message for EPLS LBL output.

• main(): Main entry point for generating EPLS LBL output.

mp_helpers.py

• processlogicblock(dfblock): Processes a block of data using logic for matching and reversals.

• worker(dfblock, outqueue): Worker function for multiprocessing, processes a block and puts results in a queue.

excel_utils.py

• openworkbook(path: str, visible: bool = False): Opens an Excel workbook, optionally visible, returns workbook and app objects.

• writedftosheetasync(...): Writes a DataFrame to an Excel sheet asynchronously.

• writeblock(start, stop): Helper for writing a block of data to Excel (used internally).

• closeworkbook(wb, appobj, save=True, usecom=False): Closes the workbook and Excel app, optionally saving changes.

• writedftosheet(...): Writes a DataFrame to a sheet synchronously.

• clearfunc(rng): Helper to clear a range in Excel (used internally).

• writedftotemplate(...): Writes a DataFrame to a template Excel file.

utils.py

• ensuredirectoryexists(path): Creates a directory if it does not exist.

• writesharedlog(scriptname, message, status="INFO"): Writes a log entry to the shared log file.

• logexception(scriptname, exc, status="ERROR"): Logs an exception with details.

• loadfilepaths(jsonfile='filepaths.json'): Loads file paths from a JSON config file.

• standardizepharmacyids(df): Standardizes pharmacy IDs in a DataFrame.

• standardizenetworkids(network): Standardizes network IDs.

• mergewithnetwork(df, network): Merges a DataFrame with network data.

• dropduplicatesdf(df): Drops duplicate rows from a DataFrame.

• cleanlogicandtier(df, logiccol='Logic', tiercol='FormularyTier'): Cleans logic and tier columns in a DataFrame.

• filterrecentdate(df, datecol='DATEFILLED'): Filters DataFrame for the most recent date.

• filterlogicandmaintenance(...): Filters data based on logic and maintenance criteria.

• filterproductsandalternative(...): Filters products and alternatives in the data.

config_loader.py

- ConfigLoader.init(self, configpath='filepaths.json'): Initializes the config loader with the given path.
- ConfigLoader.load(self): Loads configuration from the file.
- ConfigLoader.resolve(self, path): Resolves a file path from the config.
- ConfigLoader.get(self, key): Gets a config value by key.
- ConfigLoader.all(self): Returns all config values.

audit_helper.py

- makeauditentry(scriptname, message, status="INFO"): Writes an audit entry to the log, with fallback and rotation handling.

profile_runner.py

- (No explicit functions; runs profiling logic directly or via main block.)
