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
- openmdf_tier.py / openmdf_bg.py: Process Open MDF Tier and Brand/Generic logic, respectively
- tier_disruption.py / bg_disruption.py: Handle standard tier and brand/generic disruption processing
- sharx_lbl.py / epls_lbl.py: Generate SHARx and EPLS Line By Line outputs
- mp_helpers.py: Multiprocessing helpers for matching claims and reversals
- excel_utils.py: Async and safe Excel handling, using xlwings with a COM fallback
- utils.py: Common utilities (logging, filtering, ID standardization)
- config_loader.py: Centralized config management (file paths, etc.)
- audit_helper.py: Safe wrapper for audit logging
- profile_runner.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 file_paths.json and config_loader.py
- Performance Profiling: Run profile_runner.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/audit_log.csv

If OneDrive is unavailable, logs fall back to local_fallback_log.txt.

Configuration

- File Paths: Managed in file_paths.json

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

Advanced

- Performance Profiling:

```
python profile_runner.py
```

Generates profile_stats.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.
- `_build_ui()`: Constructs the GUI (buttons, progress bar, notes, etc.).
- `import_file1`, `import_file2`, `import_template_file`: File import dialogs.
- `start_process_threaded`, `start_process`, `_start_process_internal`: Launch and manage the main repricing process, including threading for responsiveness.
- `start_disruption`: Triggers disruption analysis based on user selection.
- `sharx_lbl`, `epls_lbl`: Generate SHARx/EPLS line-by-line outputs.
- `show_log_viewer`, `show_shared_log_viewer`: Display logs to the user.
- `toggle_dark_mode`: Switches between light and dark UI themes.
- `cancel_process`: Allows the user to cancel a running process.
- `write_audit_log`: Records actions/errors for audit purposes.
- `update_progress`: 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:

- openmdf_tier.py: Handles tier-based Open MDF logic.
- openmdf_bg.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:

- tier_disruption.py: For tier-based disruption analysis.

- bg_disruption.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 file_paths.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:

- `make_audit_entry`: 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 `profile_stats.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.save_default(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.apply_theme_colors(self, colors)`: Applies the selected color theme to the UI.
- `App.check_template(self, file_path)`: Validates the template file for required structure.
- `App.sharx_lbl(self)`: Generates SHARx line-by-line output from processed data.

- `App.epls_lbl(self)`: Generates EPLS line-by-line output from processed data.

- `App.show_shared_log_viewer(self)`: Displays the shared audit log to the user.

- `App._build_ui(self)`: Constructs all GUI elements (buttons, frames, labels, etc.).

- `App.import_file1(self)`: Handles importing the first data file.

- `App.import_file2(self)`: Handles importing the second data file.

- `App.import_template_file(self)`: Handles importing the Excel template file.

- `App.cancel_process(self)`: Allows the user to cancel a running process.

- `App.show_log_viewer_old(self)`: Displays the old version of the log viewer.

- `App.toggle_dark_mode(self)`: Switches the UI between dark and light mode.

- `App.update_progress(self, value=None, message=None)`: Updates the progress bar and status label.

merge.py

- `merge_files(file1_path, file2_path)`: Merges two input files, applies business logic, and outputs a formatted merged file.

openmdf_tier.py

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

openmdf_bg.py

- `process_data()`: Processes input data using Open MDF Brand/Generic logic and outputs results.

- `pivot_and_count(data)`: Helper function to pivot data and count relevant metrics.

tier_disruption.py

- `summarize_by_tier(df, col, from_val, to_val)`: Summarizes changes by tier between two values.

- `process_data()`: Runs the main tier disruption analysis and outputs results.

bg_disruption.py

- `process_data()`: Runs the main brand/generic disruption analysis and outputs results.

- `pivot(d, include_alternative=False)`: Pivots data for summary, optionally including alternatives.

- `count(d)`: Counts unique members or relevant metrics in the data.

sharx_lbl.py

- `show_message(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

- `show_message(awp, ing, total, rxs)`: Displays a summary message for EPLS LBL output.

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

mp_helpers.py

- `process_logic_block(df_block)`: Processes a block of data using logic for matching and reversals.
- `worker(df_block, out_queue)`: Worker function for multiprocessing, processes a block and puts results in a queue.

excel_utils.py

- `open_workbook(path: str, visible: bool = False)`: Opens an Excel workbook, optionally visible, returns workbook and app objects.
- `write_df_to_sheet_async(...)`: Writes a DataFrame to an Excel sheet asynchronously.
- `write_block(start, stop)`: Helper for writing a block of data to Excel (used internally).
- `close_workbook(wb, app_obj, save=True, use_com=False)`: Closes the workbook and Excel app, optionally saving changes.
- `write_df_to_sheet(...)`: Writes a DataFrame to a sheet synchronously.
- `clear_func(rng)`: Helper to clear a range in Excel (used internally).
- `write_df_to_template(...)`: Writes a DataFrame to a template Excel file.

utils.py

- `ensure_directory_exists(path)`: Creates a directory if it does not exist.
- `write_shared_log(script_name, message, status="INFO")`: Writes a log entry to the shared log file.
- `log_exception(script_name, exc, status="ERROR")`: Logs an exception with details.
- `load_file_paths(json_file='file_paths.json')`: Loads file paths from a JSON config file.
- `standardize_pharmacy_ids(df)`: Standardizes pharmacy IDs in a DataFrame.
- `standardize_network_ids(network)`: Standardizes network IDs.
- `merge_with_network(df, network)`: Merges a DataFrame with network data.
- `drop_duplicates_df(df)`: Drops duplicate rows from a DataFrame.
- `clean_logic_and_tier(df, logic_col='Logic', tier_col='FormularyTier')`: Cleans logic and tier columns in a DataFrame.
- `filter_recent_date(df, date_col='DATEFILLED')`: Filters DataFrame for the most recent date.
- `filter_logic_and_maintenance(...)`: Filters data based on logic and maintenance criteria.
- `filter_products_and_alternative(...)`: Filters products and alternatives in the data.

config_loader.py

- `ConfigLoader.__init__(self, config_path='file_paths.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

- `make_audit_entry(script_name, 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.)
