

Data Table Reshaping App



I have an idea,
please tell me if it already exists
also tell me how to implement using R or Python
or make Excel extension

basically i want to build small app that will allow you to manipulate your data table in different ways
mostly making it longer or wider or switching cols and rows

Yes — it already exists in a few forms, but there's still room to make a "small, fast, opinionated" version that feels nicer for your workflow.

What already exists (closest matches)

- **Excel: Power Query (Get & Transform)** does exactly "make longer/wider, unpivot/pivot, transpose-ish reshaping", plus joins, splits, type changes.
- **R: tidy** (`pivot_longer()`, `pivot_wider()`) and friends (`separate()`, `unite()`, `relocate()`, `rename()`).
- **Python: pandas** (`melt`, `pivot`, `pivot_table`, `stack/unstack`, `transpose`, `wide_to_long`).
- **GUI apps:** Tableau Prep / Alteryx / KNIME / OpenRefine also cover parts of this, but they're heavier.

So the concept is not new — the "product opportunity" is: a tiny app/add-in that makes the common reshapes obvious, guided, and reversible, especially for messy survey tables.

What to build (minimal useful scope)

If your app focuses on **table reshaping**, the core operations are:

1. **Longer (unpivot / melt)**
"Take multiple columns and turn them into key-value rows"
2. **Wider (pivot)**
"Turn a key column into new columns"
3. **Swap rows/cols (transpose)**
4. **Move / reorder columns**
5. **Split / combine columns (separate/unite)**
6. **Preview + undo + export** (this is what users miss in code)

A good UI pattern is: **Input table** → **choose operation** → **show preview (first 50 rows)** → **apply** → **download / copy back**.

Implementation options

Option A — R Shiny app (fastest for you if you live in R)

Why: You already use tidyverse; Shiny gives you a GUI quickly.

Core approach

- Upload or paste data (CSV/XLSX clipboard).
- Let user select:
 - ID columns (keep as-is)
 - Measure columns (to melt)
 - Names column / values column names
- Under the hood:
 - Longer: `tidyr::pivot_longer()`
 - Wider: `tidyr::pivot_wider()`
 - Transpose: `t()` (plus fix headers)
- Show preview with `DT::datatable()`
- Export: CSV/XLSX via `openxlsx`

Key design detail: store a transformation “recipe” (list of steps) so user can undo and also export the exact R code.

Option B — Python Streamlit app (also quick, more portable)

Why: Simple deployment, good UI widgets.

Core approach

- Read data with pandas (CSV/XLSX).
- UI lets user pick columns and parameters.
- Apply transformations:
 - Longer: `df.melt(id_vars=[...], value_vars=[...], var_name="...", value_name="...")`
 - Wider: `df.pivot(index=[...], columns="...", values="...")` or `pivot_table` if duplicates
 - Transpose: `df.T` (with some header handling)

Key design detail: handle duplicates safely. In “wider”, if `(index, column)` isn’t unique, you must ask for an aggregation (first/mean/sum/count) and use `pivot_table`.

Option C — Excel extension (Office Add-in) (most “real product”, but more work)

Why: Your target users may live in Excel.

Reality: This is the hardest path because it’s JavaScript/TypeScript + Office.js.

Two practical Excel routes

1. VBA / Office Scripts (quick but limited UX)

- Great for “transpose”, “reshape with assumptions”, column operations.
- Harder to build a polished multi-step wizard.

2. Office Add-in (Office.js)

- Proper sidebar UI, buttons, previews.
- You’d implement reshaping logic in JS (or call a local API).

Important note: Excel already has Power Query, so an add-in wins only if it’s dramatically simpler and focused.

A concrete “MVP spec” that’s genuinely useful

Build a Shiny or Streamlit app with just these 3 buttons:

1. Make longer

- Select ID columns
- Select columns to stack
- Choose output names (`name`, `value`)

2. Make wider

- Select index columns
- Select “column names” field
- Select “values” field
- If duplicates exist → choose aggregation

3. Transpose

- “First row becomes headers” toggle
- “Keep first column as row names” toggle

Plus:

- preview before/after
- download as xlsx/csv
- “Show code” (export the exact R/Python snippet)

This is enough to cover 70–80% of reshaping pain.

If you tell me one thing, I can give you the exact skeleton

Pick your preferred target:

- **R Shiny** (quickest for you, best if your users are analysts)
- **Python Streamlit** (easy sharing, good UI)
- **Excel add-in** (harder, but most native for non-coders)

If you say “Shiny” or “Streamlit”, I’ll reply with a ready-to-run single-file MVP that includes upload → select columns → preview → apply → export, and it will generate the reproducible code for each step.



i know about all of these like power query and R i just wanted minimal code and maximum visual UI like in spss

Got it: you want **SPSS-style “point-and-click” reshaping**, with **almost no code**, but not the full Power Query / KNIME / Alteryx universe.

Does that already exist?

Yes, partially—but each has annoyances:

- **SPSS itself**: Data → *Restructure...* does long↔wide pretty well, but it’s tied to SPSS and not as flexible for modern messy tables.
- **jamovi / JASP**: great GUI for stats, but data reshaping is limited compared to what you’re describing.
- **RStudio addins**: there are a few GUI helpers, but they’re usually single-purpose and not “SPSS-level”.
- **Power Query** is closest in Excel, but you’re saying it’s too heavy / too many steps / not the UX you want.

So your idea exists in fragments, but a **tiny dedicated “Reshape Wizard”** (longer/wider/transpose/split/unite with live preview and undo) is still a valid niche.

What to build to feel like SPSS

To match SPSS, your UI needs these elements:

1. Drag-and-drop columns into boxes

- “ID columns (keep)”
- “Columns to stack (make longer)”
- “Names column / Values column output”

2. Live preview (first 50–200 rows) updating instantly

3. Warnings in plain language

- “This ‘wider’ step will create duplicates; choose how to combine (first / mean / sum / count).”

4. **Undo / step history**
5. **Export back to Excel** (xlsx) and optionally “Copy to clipboard”
6. **Show code** (optional) just as a reproducibility button, not required

That’s basically a purpose-built, minimal “data reshape cockpit”.

Best implementation path (minimum code, maximum UI)

1) Desktop app: Python + PySide6 (Qt)

This is the closest to SPSS feel.

- UI: Qt Designer (drag-drop UI builder)
- Engine: pandas for transforms
- Extras: fast preview tables, step history, save/load “recipes”

Pros: SPSS-like, local, no browser, feels “real”

Cons: packaging/distribution effort (PyInstaller)

2) “Web app but feels like a tool”: Streamlit

Fastest to ship; UI is good enough if you do it right:

- column pickers mimic SPSS dialogs
- live preview
- download xlsx

Pros: minimal engineering, quick MVP

Cons: not true drag-and-drop; feels less “SPSS native”

3) Excel add-in: Office.js

Only do this if Excel-native is a hard requirement.

Pros: right where users are

Cons: biggest dev burden, and Power Query already competes

If your priority is “SPSS experience”, I’d do **PySide6** first.

Core reshape operations (engine logic)

Make longer (unpivot)

- user chooses:
 - ID cols
 - value cols

- output names: `variable`, `value`
- pandas: `melt`
- R: `pivot_longer`

Make wider (pivot)

- user chooses:
 - index cols
 - columns-from (creates new columns)
 - values-from
 - duplicate-handling (aggregation)
- pandas: `pivot` or `pivot_table`
- R: `pivot_wider(values_fn=...)`

Transpose

- options:
 - first row becomes headers
 - first column becomes row names

Split / Unite

- split by delimiter or regex
- combine columns with separator

What I would implement first (MVP that already feels like SPSS)

A single window with:

- Left panel: list of columns (click to select)
- Middle: operation tabs
 - Longer | Wider | Transpose | Split | Unite
- Right: "Result preview" table
- Bottom: step history + Undo + Export

And one killer feature: **"Explain in human language":**

- "You are turning 12 month columns into two columns: Month + Value, keeping RespondentID and Region unchanged."

If you want: I can give you a ready-to-run prototype immediately

Tell me which one you prefer and I'll output a complete minimal working app:

Printed using [ChatGPT to PDF](#), powered by PDFCrowd [HTML to PDF API](#).

- **A)** Streamlit MVP (fastest to try, 1 file)
- **B)** PySide6 desktop MVP (more SPSS-like)

Either way it will include:

- upload xlsx/csv
- longer + wider + transpose
- preview + undo
- export xlsx

If you pick **B (PySide6)** I'll also include packaging notes (PyInstaller) so you can send it to colleagues as a single executable.



before doing first prototypes
can we set short scope of work, what it needs to be able to do
what kinds of criterias table should be inputted, like will we be able to unmerge cells and make each row filled, and then merge back if needed? etc
can we or should we add LLM api integration for column names data structure fixing, or suggestions from it?

Scope of work for v0.1 (SPSS-like “Reshape Wizard”)

Goal: a small UI tool that reliably reshapes *rectangular* tables (tidy-ish data) with live preview and export. Not a full Excel cleaning suite.

Must-have features (v0.1)

1. Import

- CSV + XLSX (first sheet by default; allow choosing sheet)
- Detect header row (option: “Row 1 contains headers” toggle)

2. Core reshapes

- **Make longer (unpivot/melt):** choose ID cols + value cols; set output names
- **Make wider (pivot):** choose index cols + columns-from + values-from; choose aggregation if duplicates
- **Transpose:** with toggles:
 - “first row becomes headers”
 - “first column becomes row names”

3. Column operations

- Reorder columns (up/down)

- Rename columns (manual)
- Change type (text/number/date) with “convert errors to missing”

4. Preview + safety

- Before/after preview (first N rows)
- Warnings: duplicates, missing values created, type conversion failures
- Step history + Undo (at least 20 steps)

5. Export

- XLSX + CSV
- Option: “copy to clipboard as TSV” (nice for Excel pasting)

That's enough to be genuinely useful and SPSS-like.

What the input table must look like (and what we won't support in v0.1)

Supported input (v0.1)

- A **single rectangular range**: every data row has the same number of columns.
- **One header row** (or user selects which row is header).
- Cells can be blank; duplicates allowed.

Not supported in v0.1 (explicitly out of scope)

- Multi-row headers (e.g., “2024” spanning months).
- Notes/footers inside the range.
- “Report-style” Excel with formatting meaning (merged cells, subtotals, blank separator rows) as the *primary* structure.

Reason: once you go there, you're building an Excel “document parser”, not a reshape tool.

Merged cells / “unmerge and fill down” / “merge back”: should we include?

Recommendation

Include “Unmerge + fill” as an optional pre-processing step, but ONLY in a constrained form.
This is common in messy Excel (regions merged across rows), and it's solvable.

Pre-processing step: “Unmerge & fill down/right”

- For XLSX only (CSV doesn't have merges).
- Behaviour:

1. Detect merged ranges.
 2. Take the top-left value of the merged block.
 3. Fill that value into all cells in that merged block.
- Then optionally:
 - **Fill down** for selected columns (classic “Region” column blanks)

This makes the table rectangular and usable for reshaping.

“Merge back” (after reshaping)

I would **not** do “merge back” in v0.1.

- It’s not a data operation; it’s a *presentation* operation.
- Rules are ambiguous (“merge identical consecutive values?”, “merge by group?”, “max merge size?”).
- It’s doable later as an export formatting option, but it will eat time and create edge-case bugs.

Better v0.1 output: clean rectangular data; if someone wants merges, they can do it in Excel formatting afterwards.

If you really want a v0.2 “merge back” feature, define it narrowly:

- “Merge identical consecutive values in chosen columns” (grouped by other columns), max merge length, don’t merge across blank rows, etc.

Should we add LLM integration in early versions?

Recommendation: No for v0.1 (but design hooks for it)

LLMs are great for *suggestions*, but they introduce:

- Cost + latency
- Privacy/data governance concerns (especially if users load sensitive tables)
- Non-determinism (hard to test)

Where LLM is genuinely useful (v0.2+)

Add it only as an **opt-in assistant** that never changes data without user approval.

High-value LLM features:

1. Column role suggestions

- “These look like ID columns” vs “measures”
- detect likely pivot candidates

2. Rename suggestions

- clean names: spaces → underscores, transliteration, fix duplicates (**age**, **age_2**)

3. Messy-structure diagnosis

- “This sheet looks like a cross-tab report with a 2-row header; try ‘Header row = 2’ and ‘Unmerge & fill’.”

4. Generate reproducible code

- R or Python snippet that reproduces the steps

Low-value / risky LLM features (avoid)

- “Auto-fix the data structure” with silent edits
- Anything that tries to infer semantics of survey codes without explicit rules

If you do include LLM later: minimal safe design

- Default OFF
- Works on **metadata first**: column names + types + small sample (e.g., first 50 rows)
- Shows proposed actions as a checklist
- User clicks Apply

Acceptance criteria for v0.1 (so we know it’s “done”)

1. Can load a messy XLSX and produce correct results for:

- melt/unpivot
- pivot wider (with aggregation choice)
- transpose

2. Can handle:

- merged cells by “unmerge & fill”
- duplicate column names (auto-disambiguate)
- missing values

3. Provides clear warnings for:

- non-unique pivot keys
- type conversion failures

4. Exported XLSX matches preview exactly.

Suggested v0.1 vs v0.2 roadmap

v0.1 (core reshape tool)

- Import/export, longer/wider/transpose, unmerge&fill, fill-down, undo/history, warnings.

v0.2 (comfort + intelligence)

- Split/unite, regex replace, recode values, merge-back formatting (optional), templates, LLM assistant (opt-in).

If you choose the platform (desktop PySide6 vs Streamlit), I'll convert the above into a tight one-page "spec" plus a checklist of UI screens and data rules, so you can build without feature creep.



let's do Streamlit, and can you write tasks for me so that i can insert into COdex

Here's a tight, copy-pasteable task list you can drop into Codex. It's written so Codex can implement a working Streamlit MVP without you babysitting it.

text

PROJECT: Streamlit "Reshape Wizard" MVP (SPSS-like UI)

GOAL

Build a local Streamlit app that loads a table (CSV/XLSX), lets the user reshape it (Longer/Wider/Transpose), offers minimal preprocessing for messy Excel (unmerge+fill, fill-down), shows live preview, maintains step history with undo, and exports results to XLSX/CSV.

NON-GOALS (explicitly out of scope for MVP)

- Full PowerQuery-style pipeline editor
- Multi-row headers beyond "choose header row index"
- Re-merging cells on export (presentation formatting)
- LLM integrations (leave hooks, but default OFF)

TECH STACK

- Python 3.11+
- streamlit
- pandas
- openpyxl (read/write XLSX, and read merged cells ranges)
- xlsxwriter (optional for export; openpyxl OK)
- typing, dataclasses

APP STRUCTURE (suggested)

- app.py (single-file MVP is fine)
- /utils optional: io.py, transforms.py, history.py

GLOBAL REQUIREMENTS

- 1) Deterministic transformations (no randomness).
- 2) Works offline.
- 3) Uses British spelling in UI labels where applicable (e.g., "optimise", "organisation" – but keep technical words standard).
- 4) Never modify original input; transformations produce new DataFrames.
- 5) Preview first N rows (default 100) and show shape (rows, cols).
- 6) Clear warnings for pivot duplicates / aggregation choice.

DATA MODEL

- Maintain `state` in st.session_state:
 - original_df: pd.DataFrame
 - current_df: pd.DataFrame
 - history: list[dict] where each step has:
 - { "name": str,
 - "params": dict,
 - "before_shape": (r,c),
 - "after_shape": (r,c),
 - "df": pd.DataFrame (or store parquet bytes to reduce memory),
 - "timestamp": str }
 - max_history = 20

UI / PAGES (single page with tabs is fine)

- Sidebar: Import, Preprocess, Export, History
- Main area: Tabs = ["Preview", "Make longer", "Make wider", "Transpose"]
- Always show current_df preview + metadata.

TASK 1 – Project setup

- Create app.py with Streamlit layout (page config, sidebar, tabs).
- Add requirements.txt with streamlit, pandas, openpyxl, xlswriter.

TASK 2 – File import (CSV/XLSX)

Implement:

- Upload widget in sidebar.
- If CSV: `pd.read_csv` with encoding fallback (utf-8, cp1251).
- If XLSX:
 - list sheet names; allow choosing sheet.
 - allow selecting header row number (1..20). Implement by reading with `header=header_row-1`.
 - After read: normalise duplicate column names by suffixing ".2", ".3" etc.
- Set `st.session_state.original_df` and `current_df` and initialise history with step "Loaded".
- Provide basic input validation: empty file, too few columns, etc.

TASK 3 – Preprocessing: Unmerge & fill (XLSX only)

Implement function:

``unmerge_fill_xlsx(uploaded_file, sheet_name, header_row) -> pd.DataFrame``

Behaviour:

- Use `openpyxl load_workbook(data_only=True)`.
- Find merged cell ranges in the chosen sheet.
- For each merged range:
 - read top-left value
 - write that value to all cells in that range (in-memory only)
- Then extract the sheet values into a rectangular table and apply `header_row` selection.

Notes:

- Must not modify the uploaded file on disk; operate in memory.
- If user imported CSV or XLSX without merges, feature is disabled or no-op.

UI:

- Checkbox "Unmerge cells and fill values (XLSX only)" before loading OR as a preprocessing step that reloads the dataframe using workbook parsing.
- Show message: how many merged ranges were filled.

TASK 4 – Preprocessing: Fill down blanks for selected columns

Implement:

- UI multiselect of columns.
- Option: "Treat empty strings as missing".
- Apply: `df[col] = df[col].replace("", NA)` then `df[col] = df[col].ffill()`
- Push to history step "Fill down".

TASK 5 – Transform: Make longer (melt)

UI in "Make longer" tab:

- Multiselect "ID columns (keep)"
- Multiselect "Columns to stack (values)" (default = all non-ID)
- Text inputs: "Name column" default "variable", "Value column" default "value"
- Button "Apply longer"

Logic:

- Use `pd.melt` with `id_vars` and `value_vars`.
- Preserve original column order where possible.
- Push to history with params.

TASK 6 – Transform: Make wider (pivot / pivot_table)

UI in "Make wider" tab:

- Multiselect "Index columns (rows)" (default heuristic: columns with low cardinality? keep simple: empty default)
- Selectbox "Column names from" (single column)
- Selectbox "Values from" (single column)
- Detect duplicates: if `df.duplicated(subset=index_cols+[col_from]).any()`
 - show warning
 - require user to pick aggregation function: first/mean/sum/count/min/max
 - use `pivot_table` with `aggfunc` accordingly

Else:

- use `pivot`
- After pivot: flatten columns if multiindex; convert columns to string.
- Option checkbox "Reset index" default ON.
- Push to history.

TASK 7 – Transform: Transpose

UI in "Transpose" tab:

- Checkbox "First row becomes headers after transpose" default ON
- Checkbox "First column becomes row names before transpose" default OFF

- Apply transpose:
 - Optionally set index to first column then drop it
 - `df_t = df.T`
 - If first row becomes headers:
 - `new_header = df_t.iloc[0]; df_t = df_t[1:]; df_t.columns = new_header`
 - Reset index to make it rectangular; name index column "index" or user specified.
- Push to history.

TASK 8 – Preview + Metadata

Always show:

- `current_df.head(N)`
- shape, dtypes summary
- missingness summary (count of NA per column; show top 10)
- For pivot tab, show "key uniqueness check" results.

TASK 9 – History + Undo

Implement:

- In sidebar: show history steps (most recent first) with shapes.
- Buttons:
 - "Undo last step" pops last step and sets `current_df` to previous.
 - "Reset to original"
- Enforce `max_history=20` by dropping oldest (except original).

TASK 10 – Export

Sidebar export section:

- Buttons "Download CSV", "Download XLSX"
- For XLSX: use `BytesIO` + `df.to_excel(engine="openpyxl" or xlsxwriter)`.
- Ensure index is not exported unless user checkbox "Include index" ON.
- Filename based on timestamp and last operation.

TASK 11 – Quality / Edge cases

Add guards:

- Handle non-unique column names robustly.
- Handle mixed types in values column for pivot_table mean (coerce to numeric with `errors=coerce`, warn if many coerced).
- Large files: if >200k rows, warn and set preview N smaller; still allow operations.
- Ensure operations don't crash when user hasn't selected required fields; show `st.error`.

TASK 12 – Optional: Code generation (no LLM)

Add optional panel "Reproducible code":

- Show Python pandas code snippet representing the last step applied (and ideally the full pipeline list).
- Keep it simple; no external dependencies beyond pandas.

DELIVERABLE

- A working Streamlit app runnable via: ``streamlit run app.py``
- Include a short README with:
 - installation steps
 - supported inputs
 - what "Unmerge & fill" does
 - examples of Longer/Wider/Transpose

ACCEPTANCE TESTS (must pass)

- 1) Load XLSX with merged "Region" cells across rows -> unmerge+fill -> fill-down -> correct rectangular df.
- 2) Make longer on columns Jan..Dec -> output has two columns variable/value + ID columns.
- 3) Make wider where duplicates exist -> app forces aggregation selection and produces pivot_table.
- 4) Undo works across at least 5 steps.
- 5) Exported XLSX opens in Excel and matches preview.

If you want, I can also add a **small synthetic test dataset generator task** (creates a merged-cell XLSX + a wide-to-long sample) so you can validate the MVP quickly.



can you make me a small fake table to test

Yep — here are **two tiny fake tables** that are perfect for testing your app:

1. one “messy Excel-style” (for unmerge/fill-down)
2. one “reshape playground” (for longer/wider/transpose)

You can copy-paste these into Excel or save as CSV.




1 Messy table (simulates merged cells + fill-down need)

This mimics a sheet where *Region* and *School* were merged in Excel.

```
javascript
```

```
Region,School,Student,Math,English
North,North High,Alice,78,85
,,Bob,67,73
,,Charlie,90,88
South,South High,David,55,60
,,Eva,72,70
```

What this tests:

-  Fill-down on **Region** and **School**
-  Melt Math/English to long
-  Pivot back to wide if needed

Expected after fill-down:

```
North,North High,Alice,78,85
North,North High,Bob,67,73
North,North High,Charlie,90,88
South,South High,David,55,60
South,South High,Eva,72,70
```

2 Wide table for longer → wider testing

```
r
```

```
ID,Gender,Jan,Feb,Mar
1,M,10,12,9
2,F,8,11,13
3,F,7,6,5
```

Test cases:

Make longer (melt):

- ID columns: **ID**, **Gender**

- Value columns: Jan, Feb, Mar

Expected:

```
r
ID,Gender,Month,Value
1,M,Jan,10
1,M,Feb,12
1,M,Mar,9
2,F,Jan,8
2,F,Feb,11
2,F,Mar,13
3,F,Jan,7
3,F,Feb,6
3,F,Mar,5
```

3 Table that forces aggregation in “Make wider”

```
javascript
Person,Year,ScoreType,Score
Alice,2023,Math,80
Alice,2023,Math,85
Alice,2023,English,90
Bob,2023,Math,70
Bob,2023,English,75
```

If user tries:

- index = Person, Year
- columns-from = ScoreType
- values-from = Score

Your app should:

⚠ detect duplicates for (Alice,2023,Math)
and force aggregation:

- mean → Math = 82.5
- first → Math = 80
- max → Math = 85

Expected with mean:

```
javascript
Person,Year,English,Math
Alice,2023,90,82.5
Bob,2023,75,70
```

4 Tiny transpose test

CSS

```
Var,A,B,C
v1,1,2,3
v2,4,5,6
```

Transpose with:

- ✓ first column = row names
- ✓ first row = headers

Expected:

CSS

```
Var,v1,v2
A,1,4
B,2,5
C,3,6
```

If you want, I can also generate for you:

- ✓ an actual .xlsx file with **real merged cells**
- ✓ and a .csv bundle of all test cases

so you can drop them straight into your Streamlit app.

Just say:

"Make me a test Excel file with merged cells."



make it as xlsx file with different sheets

python

```
from openpyxl import Workbook

wb = Workbook()

# Sheet 1: Messy merged cells
ws1 = wb.active
ws1.title = "Messy_Merged"
ws1.append(["Region", "School", "Student", "Math", "English"])
ws1.append(["North", "North High", "Alice", 78, 85])
ws1.append(["", "", "Bob", 67, 73])
ws1.append(["", "", "Charlie", 90, 88])
ws1.append(["South", "South High", "David", 55, 60])
ws1.append(["", "", "Eva", 72, 70])

# Merge cells for Region and School
ws1.merge_cells("A2:A4")
ws1.merge_cells("B2:B4")
ws1.merge_cells("A5:A6")
ws1.merge_cells("B5:B6")

# Sheet 2: Wide table
ws2 = wb.create_sheet("Wide_for_Longer")
ws2.append(["ID", "Gender", "Jan", "Feb", "Mar"])
ws2.append([1, "M", 10, 12, 9])
```



```

ws2.append([2, "F", 8, 11, 13])
ws2.append([3, "F", 7, 6, 5])

# Sheet 3: Needs aggregation
ws3 = wb.create_sheet("Needs_Aggregation")
ws3.append(["Person", "Year", "ScoreType", "Score"])
ws3.append(["Alice", 2023, "Math", 80])
ws3.append(["Alice", 2023, "Math", 85])
ws3.append(["Alice", 2023, "English", 90])
ws3.append(["Bob", 2023, "Math", 70])
ws3.append(["Bob", 2023, "English", 75])

# Sheet 4: Transpose test
ws4 = wb.create_sheet("Transpose_Test")
ws4.append(["Var", "A", "B", "C"])
ws4.append(["v1", 1, 2, 3])
ws4.append(["v2", 4, 5, 6])

file_path = "/mnt/data/reshape_test_tables.xlsx"
wb.save(file_path)

file_path

```

Result
'/mnt/data/reshape_test_tables.xlsx'

Done. I created an **Excel file with multiple sheets**, including real merged cells for testing your unmerge+fill logic.

Sheets inside:

1. **Messy_Merged** – Region & School are truly merged
2. **Wide_for_Longer** – for melt/unpivot
3. **Needs_Aggregation** – forces pivot aggregation
4. **Transpose_Test** – for transpose logic

You can download it here:

reshape_test_tables.xlsx

	A	B	C	D	E	F	G
1	Region	School	Student	Math	English		
2	North	North High	Alice	78	85		
3			Bob	67	73		
4			Charlie	90	88		
5	South	South High	David	55	60		
6			Eva	72	70		
7							
8							
9							
10							
11							
12							
13							
14							

This file is perfect to validate:

- unmerge + fill-down

- longer
- wider (with duplicate detection)
- transpose

If you want, next I can:

- generate a second file with multi-row headers (for future v0.2), or
- help you write the first Streamlit loader that correctly handles this file.



i want to write instructon that i want to be able to see preview of pivot wider longer and any other operation for codex

Use this as a Codex task/spec block (copy-paste). It's focused specifically on **preview-before-apply** for Longer/Wider/Transpose (and any future ops).

text

FEATURE: Preview mode for all transformations (SPSS-like)

USER STORY

As a user, before applying any operation (pivot wider, pivot longer, transpose, fill-down, unmerge+fill, etc.),
I want to see a live preview of the resulting table (first N rows) and key diagnostics (row/col counts, warnings),
so I can verify the effect before committing changes.

DEFINITION OF DONE

- Every operation screen shows BOTH:
 - 1) Current table preview (before)
 - 2) Preview result (after) based on current UI parameters, without changing current_df
- "Apply" only commits after preview is computed successfully.
- If parameters are incomplete/invalid, preview panel shows a friendly error instead of crashing.
- Preview updates automatically when parameters change (with debounce / manual "Update preview" button for heavy ops).
- Preview uses only first N rows for display, but shape/warnings should be computed on full data where feasible.

STATE CHANGES

- Add to st.session_state:
 - preview_df: pd.DataFrame | None
 - preview_meta: dict | None (before_shape, after_shape, warnings list, notes)
 - preview_error: str | None
 - preview_params_hash: str (so we can avoid recomputing if no changes)
 - preview_limit: int (default 100)
- IMPORTANT: preview does NOT modify current_df or history.

UI REQUIREMENTS (apply to each operation tab)

- 1) "Before" panel
 - show current_df.head(preview_limit)
 - show current_df shape + dtypes summary snippet
- 2) "After (Preview)" panel
 - show preview_df.head(preview_limit)
 - show after shape
 - show warnings/notes in plain language
- 3) Buttons / controls
 - "Update preview" button (always available)
 - "Apply operation" button (enabled only if preview computed and no fatal errors)
 - Optional checkbox "Auto-update preview on parameter change" (default ON; if OFF, require Update preview)

4) Performance:

- wrap preview computation in st.spinner
- use st.cache_data for preview computations keyed by (df fingerprint + operation + params)
- for very large dfs (>200k rows), auto-disable auto-update and require button click.

IMPLEMENTATION DETAIL – Generic preview engine

Create a generic function:

```
compute_preview(
    df: pd.DataFrame,
    op_name: str,
    params: dict
) -> tuple[pd.DataFrame, dict]
```

Where meta dict includes:

- before_shape, after_shape
- warnings: list[str]
- diagnostics: dict (e.g., duplicates_detected, agg_required, missing_created, type_coercions)

Each operation provides:

- validate_params(params, df) -> (ok: bool, error_msg: str|None)
- transform(df, params) -> df_out + operation_warnings

PREVIEW RULES FOR SPECIFIC OPS

A) Make longer (melt)

Params:

- id_cols: list[str]
- value_cols: list[str]
- var_name: str
- value_name: str

Preview behaviour:

- Validate value_cols non-empty, no overlap issues, var/value names not empty.
- Compute df_out = df.melt(...)
- Warnings:
 - if value_cols empty -> fatal
 - if id_cols empty -> note "No ID columns; all columns stacked"
 - if resulting rows huge (>5M) -> warning and still allow preview but note it may be slow on apply

B) Make wider (pivot / pivot_table)

Params:

- index_cols: list[str]
- columns_from: str
- values_from: str
- aggfunc: str (required only if duplicates exist)
- reset_index: bool

Preview behaviour:

- Validate columns selected.
- Compute duplicates flag on full df:


```
dup = df.duplicated(subset=index_cols + [columns_from]).any()
```
- If dup is True:
 - require aggfunc (first/mean/sum/count/min/max)
 - show warning explaining duplicates and chosen aggregation
 - use pivot_table
- Else:
 - use pivot
- Flatten MultiIndex columns; ensure column names are strings.
- Warnings:
 - if mean chosen and values_from non-numeric -> coerce numeric with errors=coerce, warn about coerced count
 - if resulting columns count > 500 -> warning "Very wide table"

C) Transpose

Params:

- first_col_as_index: bool
- first_row_as_headers: bool
- index_col_name: str (default "index")

Preview behaviour:

- Perform transpose and optional header handling.
- Warnings:
 - if headers after transpose are not unique -> auto-disambiguate and warn

D) Preprocess steps (fill-down, unmerge+fill)

- Same preview pattern: show before and after without commit.
- For fill-down: show how many blanks were filled per column.

COMMIT / APPLY BEHAVIOUR

- Apply button triggers the same transform function used for preview.
- After apply:
 - push to history with name, params, before/after shapes, warnings
 - set `current_df = result`
 - clear `preview_df/meta/error`

ERROR HANDLING

- Any exception in preview should be caught; set `preview_error` and show `st.error` with human-readable message.
- Never break the app; no stack traces in UI unless debug mode.

TESTS (manual acceptance)

- For each op tab: changing a parameter updates preview; pressing Apply changes `current_df` and history.
- Pivot wider: duplicates detection forces agg selection and preview warns.
- Undo after applying returns to prior table.
- Preview never changes `current_df` unless Apply clicked.

If you want, I can also rewrite that into a **shorter “Codex-friendly”** version (some Codex runs do better with less text), but the above is the complete behaviour spec.



is it possible to add option like insert pasted table from excel? or too difficult? make short answer

Short answer: **yes, it’s possible and not too difficult.**

In Streamlit you can:

- Add a **“Paste from clipboard” text area**
- User pastes Excel cells (tab-separated)
- You parse it with `pd.read_csv(io.StringIO(text), sep="\t")`

Limitations:

- You won’t get formatting or merged cells (only values)
- But for quick reshaping, it works very well

So:

- ✓ feasible
- ✓ low effort
- ⚠ values only (no merges, no styles)

It’s a good v0.1 feature.



write me a task for codex to implement this

Here is a clean **Codex-ready task** you can copy-paste:

text

FEATURE: Paste table from Excel (clipboard input)

USER STORY

As a user, I want to paste a table copied from Excel directly into the app (Ctrl+V), so I can reshape data without saving a file first.

SCOPE

- Support pasting rectangular tables copied from Excel or Google Sheets.
- Values only (no formatting, no merged cells).
- Treated the same as a loaded CSV/XLSX after parsing.

UI

- Add a section in the sidebar: "Paste table from Excel"
- Provide:
 - st.text_area(label="Paste your table here (Ctrl+V)")
 - Button: "Load pasted table"
- When loaded, it replaces current_df and resets history.

IMPLEMENTATION

- On button click:
 - Read text from text_area
 - Parse using:
 - pd.read_csv(StringIO(text), sep="\t")
 - Trim empty trailing columns/rows
 - Auto-generate column names if missing
 - Normalise duplicate column names (suffix .2, .3, ...)

VALIDATION

- If parsing fails → show st.error("Could not parse pasted table. Make sure it is copied from Excel as a table.")
- If only 1 column detected → warn user it may not be tab-separated.

STATE

- Set:
 - st.session_state.original_df
 - st.session_state.current_df
 - reset history with step "Loaded from clipboard"

EDGE CASES

- Strip leading/trailing whitespace
- Treat empty strings as NA
- Limit max pasted size (e.g., 200k rows) with warning

ACCEPTANCE TEST

- 1) Copy cells from Excel (A1:D5), paste → table loads correctly.
- 2) Columns and row count match Excel.
- 3) After paste, all reshape operations work normally.
- 4) Undo/Reset works after paste.

If you want, I can next:

- ✓ merge this into your big task list
- ✓ or generate the exact Python code for this feature
- ✓ or adapt it to also support semicolon-separated pastes (European Excel)



there is also one behaviour that i wanna fix somehow, when table has some empty colnames it's impossible to manipulate them, can we add automatically colnames, or ask user fix them, and also add option rename any column name

prepare task for codex please,
thank you

Here is a **Codex-ready task** for handling empty column names + renaming:

text

FEATURE: Handle empty column names + allow renaming columns

USER STORY

As a user, when my table has empty or duplicate column names,
I want the app to automatically fix them or ask me to fix them,
so I can select and manipulate all columns in reshape operations.

SCOPE

- Detect empty, missing, or duplicate column names after load (file or paste).
- Auto-generate safe column names when needed.
- Provide UI to manually rename any column at any time.

DETECTION RULES

On every load (CSV/XLSX/paste):

- If any column name is:
 - empty string ""
 - NaN
 - only whitespace→ treat as invalid.
- If duplicate column names exist → treat as invalid.

AUTO-FIX BEHAVIOUR (default)

- Replace invalid names with:
Column_1, Column_2, Column_3, ...
(only for those that are invalid)
- For duplicates:
name, name.2, name.3, ...
- Show info message:
"Some columns had empty or duplicate names and were auto-renamed."

OPTIONAL USER CONFIRMATION MODE

(If enabled via checkbox "Ask before fixing column names"):

- Show table:
Old name | New suggested name
- User can edit new names before applying.

RENAME UI (always available)

Add a section: "Rename columns"

- Show a table or list:
Current name | Text input (new name)
- Button: "Apply renames"
- Validate:
 - No empty names
 - No duplicates
 - Strip leading/trailing spaces
- If invalid → st.error with clear message.

STATE HANDLING

- Renaming is a transformation:
 - Push to history as step "Rename columns"
 - Update current_df
- Renaming should update:
 - All previews
 - All operation selectors (ID cols, value cols, etc.)

EDGE CASES

- If all columns are empty → generate Column_1..Column_n.
- If renaming causes duplicates → block and show error.
- Preserve original order of columns.

ACCEPTANCE TESTS

- 1) Load table with headers: "", "", "Value" → auto becomes Column_1, Column_2, Value.
- 2) Load table with duplicate headers: A, A, B → becomes A, A.2, B.

- 3) User renames Column_1 → Region → works and appears in selectors.
- 4) User tries to rename two columns to same name → blocked with error.
- 5) After renaming, pivot/melt operations work normally.

If you want, I can:

- ✓ merge this into your big Codex task list
- ✓ or write the exact Python functions (`normalise_column_names`, `rename_columns_ui`)
- ✓ or prepare a UI mock layout for this part

Just say which one.



"Ask before fixing column names" doesnt work
 empty columns from excel are just names "Unnamed: 1" etc
 Rename columns doesn't work
 when i paste from excel empty columns remain being empty names, without even "Unnamed: 1"
 etc
 write tasks to fix

this is code

"""

Reshape Wizard MVP with preview-first UX.

- Load CSV/XLSX, optional unmerge+fill, fill-down, longer/wider/transpose.
- Every operation offers preview (before/after) before committing.

"""

from __future__ import annotations

import json

from datetime import datetime

from io import BytesIO, StringIO

from typing import Any, Dict, List, Optional, Tuple

import pandas as pd

import streamlit as st

from openpyxl import load_workbook

PREVIEW_DEFAULT = 100

LARGE_ROW_THRESHOLD = 200_000

VERY_WIDE_THRESHOLD = 500

HUGE_ROW_WARNING = 5_000_000

def init_state() -> None:

"""Ensure session_state keys exist."""

ss = st.session_state

ss.setdefault("original_df", None)

```

ss.setdefault("current_df", None)
ss.setdefault("history", [])
ss.setdefault("max_history", 20)
ss.setdefault("last_merge_fill", None)
ss.setdefault("preview_n", PREVIEW_DEFAULT)
ss.setdefault("preview_df", None)
ss.setdefault("preview_meta", None)
ss.setdefault("preview_error", None)
ss.setdefault("preview_params_hash", None)
ss.setdefault("preview_limit", PREVIEW_DEFAULT)
ss.setdefault("upload_cache", {}) # {"bytes":..., "name":..., "ext":..., "sheets": [...]}

```

```

def normalise_duplicate_columns(cols) -> List[str]:
    """Suffix duplicate column names with .2, .3 ..."""
    counts: Dict[str, int] = {}
    new_cols: List[str] = []
    for col in cols:
        base = "Unnamed" if pd.isna(col) else str(col)
        counts[base] = counts.get(base, 0) + 1
        new_cols.append(base if counts[base] == 1 else f"{base}.{counts[base]}")
    return new_cols

```

```

def is_invalid_colname(name: Any) -> bool:
    return pd.isna(name) or (isinstance(name, str) and name.strip() == "")

```

```

def sanitise_columns(columns: List[Any]) -> Tuple[List[str], bool]:
    """Return safe column names and whether any fixes were applied."""
    provisional = []
    fixed = False
    for idx, col in enumerate(columns, start=1):
        if is_invalid_colname(col):
            provisional.append(f"Column_{idx}")
            fixed = True
        else:
            provisional.append(str(col).strip())
    # Handle duplicates
    deduped = normalise_duplicate_columns(provisional)
    if deduped != provisional:
        fixed = True
    return deduped, fixed

```

```

def add_history_step(name: str, params: Dict[str, Any], df: pd.DataFrame, before_shape: Tuple[int, int]) -> None:
    """Record a transformation; keep max_history."""
    step = {
        "name": name,
        "params": params,

```



```

    "before_shape": before_shape,
    "after_shape": df.shape,
    "df": df.copy(deep=True),
    "timestamp": datetime.now().isoformat(timespec="seconds"),
}
st.session_state.history.append(step)
if len(st.session_state.history) > st.session_state.max_history:
    st.session_state.history.pop(0)

def clear_preview() -> None:
    st.session_state.preview_df = None
    st.session_state.preview_meta = None
    st.session_state.preview_error = None
    st.session_state.preview_params_hash = None

def get_sheet_names(file_bytes: bytes) -> List[str]:
    wb = load_workbook(BytesIO(file_bytes), read_only=True, data_only=True)
    return wb.sheetnames

def read_csv_with_fallback(file_bytes: bytes) -> pd.DataFrame:
    for enc in ("utf-8", "cp1251"):
        try:
            df = pd.read_csv(BytesIO(file_bytes), encoding=enc)
            df.columns = normalise_duplicate_columns(df.columns)
            return df
        except Exception:
            continue
    raise ValueError("Could not read CSV with utf-8 or cp1251 encodings.")

def unmerge_fill_xlsx(file_bytes: bytes, sheet_name: str, header_row: int) -> Tuple[pd.DataFrame,
int]:
    """Fill merged cell ranges with their top-left value and return DataFrame."""
    wb = load_workbook(BytesIO(file_bytes), data_only=True)
    if sheet_name not in wb.sheetnames:
        raise ValueError(f"Sheet '{sheet_name}' not found.")
    ws = wb[sheet_name]

    filled_ranges = 0
    for merged_range in list(ws.merged_cells.ranges):
        min_col, min_row, max_col, max_row = merged_range.bounds
        top_left = ws.cell(row=min_row, column=min_col).value
        if top_left is None:
            ws.unmerge_cells(str(merged_range))
            continue
        for row in range(min_row, max_row + 1):
            for col in range(min_col, max_col + 1):
                ws.cell(row=row, column=col).value = top_left

```

```

ws.unmerge_cells(str(merged_range))
filled_ranges += 1

data = [list(row) for row in ws.iter_rows(values_only=True)]
if not data:
    return pd.DataFrame(), filled_ranges

header_idx = max(header_row - 1, 0)
if header_idx >= len(data):
    raise ValueError("Header row is beyond the data range.")

header = data[header_idx]
records = data[header_idx + 1 :]
df = pd.DataFrame(records, columns=header)
df.columns = normalise_duplicate_columns(df.columns)
return df, filled_ranges

def load_xlsx_standard(file_bytes: bytes, sheet_name: str, header_row: int) -> pd.DataFrame:
    df = pd.read_excel(BytesIO(file_bytes), sheet_name=sheet_name, header=header_row - 1,
engine="openpyxl")
    df.columns = normalise_duplicate_columns(df.columns)
    return df

def load_dataframe(
    file_bytes: bytes,
    filename: str,
    sheet_name: Optional[str],
    header_row: int,
    unmerge_fill: bool,
) -> Tuple[pd.DataFrame, Optional[int]]:
    ext = filename.lower().split(".")[1]
    merge_info: Optional[int] = None
    if ext == "csv":
        df = read_csv_with_fallback(file_bytes)
    elif ext in ("xlsx", "xlsm", "xls"):
        if sheet_name is None:
            raise ValueError("Please choose a sheet.")
        if unmerge_fill:
            df, merge_info = unmerge_fill_xlsx(file_bytes, sheet_name, header_row)
        else:
            df = load_xlsx_standard(file_bytes, sheet_name, header_row)
    else:
        raise ValueError("Unsupported file type. Please upload CSV or XLSX.")

    if df.empty or df.shape[1] < 1:
        raise ValueError("Loaded data is empty or has no columns.")
    return df, merge_info

```

```
def load_from_clipboard(text: str) -> pd.DataFrame:
    """Parse tab-separated clipboard text from Excel/Sheets."""
    cleaned = text.strip()
    if not cleaned:
        raise ValueError("No data pasted.")
    try:
        df = pd.read_csv(StringIO(cleaned), sep="\t", dtype=str)
    except Exception as exc:
        raise ValueError(
            "Could not parse pasted table. Make sure it is copied from Excel as a table."
        ) from exc
    df = df.replace("", pd.NA)
    df = df.dropna(axis=0, how="all").dropna(axis=1, how="all")
    if df.empty or df.shape[1] == 0:
        raise ValueError("Pasted table is empty after trimming blank rows/columns.")

    safe_cols, fixed = sanitise_columns(list(df.columns))
    df.columns = safe_cols
    return df


def fill_down(df: pd.DataFrame, columns: List[str], treat_empty_as_missing: bool) ->
    Tuple[pd.DataFrame, Dict[str, int]]:
    """Fill down selected columns; return df and counts per column."""
    if not columns:
        return df, {}
    filled = df.copy()
    filled_counts: Dict[str, int] = {}
    for col in columns:
        before_na = filled[col].isna().sum()
        if treat_empty_as_missing:
            filled[col] = filled[col].replace("", pd.NA)
        filled[col] = filled[col].ffill()
        after_na = filled[col].isna().sum()
        filled_counts[col] = int(before_na - after_na)
    return filled, filled_counts


def make_longer(
    df: pd.DataFrame,
    id_cols: List[str],
    value_cols: List[str],
    name_col: str,
    value_col: str,
) -> pd.DataFrame:
    return pd.melt(df, id_vars=id_cols, value_vars=value_cols, var_name=name_col,
        value_name=value_col)


def make_wider(
    df: pd.DataFrame,
```

```

index_cols: List[str],
column_source: str,
value_source: str,
aggfunc: Optional[str],
reset_index: bool,
) -> pd.DataFrame:
    work_df = df.copy()
    if aggfunc == "mean":
        work_df[value_source] = pd.to_numeric(work_df[value_source], errors="coerce")

    if aggfunc:
        widened = work_df.pivot_table(
            index=index_cols,
            columns=column_source,
            values=value_source,
            aggfunc=aggfunc,
        )
    else:
        widened = work_df.pivot(index=index_cols, columns=column_source, values=value_source)

    if isinstance(widened.columns, pd.MultiIndex):
        widened.columns = ["_".join(map(str, col)).strip() for col in widened.columns.values]
    else:
        widened.columns = [str(c) for c in widened.columns]

    if reset_index:
        widened = widened.reset_index()

    return widened


def transpose_df(
    df: pd.DataFrame,
    use_first_row_as_header: bool,
    use_first_col_as_index: bool,
) -> Tuple[pd.DataFrame, bool]:
    temp = df.copy()
    if use_first_col_as_index and not temp.empty:
        temp = temp.set_index(temp.columns[0])
    transposed = temp.transpose()

    header_adjusted = False
    if use_first_row_as_header and not transposed.empty:
        header = transposed.iloc[0]
        transposed = transposed.iloc[1:]
        transposed.columns = header
        header_adjusted = True

    transposed = transposed.reset_index().rename(columns={"index": "index"})
    before_cols = list(transposed.columns)
    transposed.columns = normalise_duplicate_columns(transposed.columns)

```

```
warnings_adjusted = before_cols != list(transposed.columns)
return transposed, header_adjusted or warnings_adjusted

def preview_rows(df: pd.DataFrame) -> int:
    limit = st.session_state.get("preview_limit", PREVIEW_DEFAULT)
    if df.shape[0] > LARGE_ROW_THRESHOLD:
        return min(50, df.shape[0], limit)
    return min(limit, df.shape[0])

def df_fingerprint(df: pd.DataFrame) -> str:
    """Lightweight fingerprint to key cache."""
    sample = df.head(50)
    values_hash = pd.util.hash_pandas_object(sample, index=True).sum()
    return f"{df.shape}-{hash(tuple(sample.columns))}-{int(values_hash)}"

def show_before_after(
    before_df: Optional[pd.DataFrame],
    after_df: Optional[pd.DataFrame],
    meta: Optional[Dict[str, Any]],
    error: Optional[str],
    title_after: str,
) -> None:
    col_before, col_after = st.columns(2)
    with col_before:
        if before_df is None:
            st.info("Load data to see a preview.")
        else:
            n = preview_rows(before_df)
            st.caption(f"Before • {before_df.shape[0]} rows × {before_df.shape[1]} columns (showing {n})")
            st.dataframe(before_df.head(n))
            st.caption("Dtypes")
            st.write(before_df.dtypes.to_frame("dtype"))
    with col_after:
        st.caption(title_after)
        if error:
            st.error(error)
            return
        if after_df is None or meta is None:
            st.info("Update preview to see the result.")
            return
        n = preview_rows(after_df)
        st.caption(f"After • {meta['after_shape'][0]} rows × {meta['after_shape'][1]} columns (showing {n})")
        st.dataframe(after_df.head(n))
        warnings = meta.get("warnings", [])
        diagnostics = meta.get("diagnostics", {})
        if warnings:
```

```

        st.warning("\n".join(warnings))
    if diagnostics:
        st.caption(f"Diagnostics: {diagnostics}")

@st.cache_data(show_spinner=False, hash_funcs={pd.DataFrame: lambda _: 0})
def cached_preview(df_hash: str, op_name: str, params_json: str, df: pd.DataFrame) ->
    Tuple[pd.DataFrame, Dict[str, Any]]:
    params = json.loads(params_json)
    return _compute_preview(df, op_name, params)

def _compute_preview(df: pd.DataFrame, op_name: str, params: Dict[str, Any]) ->
    Tuple[pd.DataFrame, Dict[str, Any]]:
    before_shape = df.shape
    warnings: List[str] = []
    diagnostics: Dict[str, Any] = {}

    if op_name == "fill_down":
        columns = params.get("columns", [])
        treat_empty = params.get("treat_empty", True)
        if not columns:
            raise ValueError("Select at least one column to fill down.")
        df_out, counts = fill_down(df, columns, treat_empty)
        diagnostics["filled_counts"] = counts
        return df_out, {"before_shape": before_shape, "after_shape": df_out.shape, "warnings":
warnings, "diagnostics": diagnostics}

    if op_name == "unmerge_fill":
        cache = st.session_state.upload_cache
        if not cache.get("bytes") or cache.get("name") != params.get("cache_key"):
            raise ValueError("No workbook available for unmerge preview. Reload the file first.")
        df_out, filled = unmerge_fill_xlsx(cache["bytes"], params["sheet_name"],
params["header_row"])
        safe_cols, fixed = sanitise_columns(list(df_out.columns))
        df_out.columns = safe_cols
        if fixed:
            warnings.append("Column names were auto-renamed after unmerge.")
            warnings.append(f"Filled {filled} merged ranges.")
            diagnostics["filled_ranges"] = filled
            return df_out, {"before_shape": before_shape, "after_shape": df_out.shape, "warnings":
warnings, "diagnostics": diagnostics}

    if op_name == "longer":
        id_cols = params.get("id_cols", [])
        value_cols = params.get("value_cols", [])
        name_col = params.get("name_col") or "variable"
        value_col = params.get("value_col") or "value"
        if not value_cols:
            raise ValueError("Select at least one value column.")
        for col in id_cols + value_cols:

```

```

        if col not in df.columns:
            raise ValueError(f"Column '{col}' not found.")
    df_out = make_longer(df, id_cols, value_cols, name_col, value_col)
    if not id_cols:
        warnings.append("No ID columns selected; all columns stacked.")
    expected_rows = len(df) * len(value_cols)
    if expected_rows > HUGE_ROW_WARNING:
        warnings.append("Result is very large; applying may be slow.")
    return df_out, {"before_shape": before_shape, "after_shape": df_out.shape, "warnings":
warnings, "diagnostics": diagnostics}

if op_name == "wider":
    index_cols = params.get("index_cols", [])
    column_source = params.get("column_source")
    value_source = params.get("value_source")
    aggfunc = params.get("aggfunc")
    reset_index = params.get("reset_index", True)
    if not column_source or not value_source:
        raise ValueError("Choose 'column names from' and 'values from'.")
    if column_source == value_source:
        raise ValueError("Column source and value source must differ.")
    for col in index_cols + [column_source, value_source]:
        if col not in df.columns:
            raise ValueError(f"Column '{col}' not found.")
    duplicates = df.duplicated(subset=index_cols + [column_source]).any()
    diagnostics["duplicates_detected"] = bool(duplicates)
    if duplicates and not aggfunc:
        raise ValueError("Duplicates detected; choose an aggregation.")
    coercions = 0
    if aggfunc == "mean":
        series = pd.to_numeric(df[value_source], errors="coerce")
        coercions = int(series.isna().sum() - df[value_source].isna().sum())
        df_temp = df.copy()
        df_temp[value_source] = series
        df_out = make_wider(df_temp, index_cols, column_source, value_source, aggfunc,
reset_index)
    else:
        df_out = make_wider(df, index_cols, column_source, value_source, aggfunc, reset_index)
    if duplicates:
        warnings.append(f"Duplicates found; aggregation='{aggfunc}' applied.")
    if coercions > 0:
        warnings.append(f"Coerced {coercions} values to numeric for mean.")
        diagnostics["coercions"] = coercions
    if df_out.shape[1] > VERY_WIDE_THRESHOLD:
        warnings.append("Resulting table is very wide.")
    return df_out, {"before_shape": before_shape, "after_shape": df_out.shape, "warnings":
warnings, "diagnostics": diagnostics}

if op_name == "transpose":
    use_first_row = params.get("first_row_header", True)
    use_first_col = params.get("first_col_index", False)

```

```
df_out, adjusted = transpose_df(df, use_first_row, use_first_col)
if adjusted:
    warnings.append("Headers were adjusted/normalised after transpose.")
    return df_out, {"before_shape": before_shape, "after_shape": df_out.shape, "warnings":
warnings, "diagnostics": diagnostics}
```

```
raise ValueError(f"Unknown operation '{op_name}'.")
```

```
def trigger_preview(df: pd.DataFrame, op_name: str, params: Dict[str, Any], auto_update: bool,
params_hash: str, cache_df_hash: str) -> None:
```

```
    """Compute and store preview if needed."""
```

```
    do_update = auto_update and st.session_state.preview_params_hash != params_hash
```

```
    button_clicked = st.button("Update preview", key=f"update_{op_name}")
```

```
    if button_clicked:
```

```
        do_update = True
```

```
    if not do_update:
```

```
        return
```

```
    with st.spinner("Computing preview..."):
```

```
        try:
```

```
            preview_df, meta = cached_preview(cache_df_hash, op_name, json.dumps(params,
sort_keys=True), df)
```

```
            st.session_state.preview_df = preview_df
```

```
            st.session_state.preview_meta = meta
```

```
            st.session_state.preview_error = None
```

```
            st.session_state.preview_params_hash = params_hash
```

```
        except Exception as exc:
```

```
            st.session_state.preview_df = None
```

```
            st.session_state.preview_meta = None
```

```
            st.session_state.preview_error = str(exc)
```

```
            st.session_state.preview_params_hash = params_hash
```

```
def get_preview_state(params_hash: str) -> Tuple[Optional[pd.DataFrame], Optional[Dict[str, Any]],
Optional[str]]:
```

```
    """Return preview data only if it matches the provided hash."""
```

```
    if st.session_state.preview_params_hash == params_hash:
```

```
        return st.session_state.preview_df, st.session_state.preview_meta,
```

```
st.session_state.preview_error
```

```
    return None, None, None
```

```
def render_history_sidebar() -> None:
```

```
    st.sidebar.subheader("History")
```

```
    history = list(reversed(st.session_state.history))
```

```
    if not history:
```

```
        st.sidebar.caption("No steps yet.")
```

```
    return
```

```
    for step in history:
```

```
        st.sidebar.write(
```



```

        f"{step['timestamp']} — {step['name']} "
        f"{step['before_shape']} → {step['after_shape']}"
    )

col1, col2 = st.sidebar.columns(2)
if col1.button("Undo last step", use_container_width=True):
    if len(st.session_state.history) > 1:
        st.session_state.history.pop()
        st.session_state.current_df = st.session_state.history[-1]["df"].copy()
        clear_preview()
    else:
        st.sidebar.warning("Nothing to undo.")

if col2.button("Reset to original", use_container_width=True):
    if st.session_state.original_df is not None:
        st.session_state.current_df = st.session_state.original_df.copy()
        st.session_state.history = st.session_state.history[:1]
        clear_preview()
    else:
        st.sidebar.warning("No original data loaded.")

def export_buttons(df: pd.DataFrame) -> None:
    if df is None:
        st.sidebar.caption("Load data to enable export.")
        return
    st.sidebar.subheader("Export")
    include_index = st.sidebar.checkbox("Include index", value=False)
    timestamp = datetime.now().strftime("%Y%m%d-%H%M%S")
    last_step = st.session_state.history[-1]["name"] if st.session_state.history else "export"

    csv_data = df.to_csv(index=include_index).encode("utf-8")
    st.sidebar.download_button(
        label="Download CSV",
        data=csv_data,
        file_name=f"reshape_{last_step}_{timestamp}.csv",
        mime="text/csv",
    )

    buffer = BytesIO()
    df.to_excel(buffer, index=include_index, engine="openpyxl")
    st.sidebar.download_button(
        label="Download XLSX",
        data=buffer.getvalue(),
        file_name=f"reshape_{last_step}_{timestamp}.xlsx",
        mime="application/vnd.openxmlformats-officedocument.spreadsheetml.sheet",
    )

def code_snippet_for_last_step() -> str:
    if not st.session_state.history:

```

```

    return "# No steps yet"
step = st.session_state.history[-1]
name = step["name"]
params = step["params"]
code_map = {
    "Loaded": f"df = pd.read_{params.get('source', 'csv')}(...)",
    "Fill down": (
        "for col in columns:\n"
        "    df[col] = df[col].replace(", pd.NA)\n"
        "    df[col] = df[col].ffill()"
    ),
    "Make longer": (
        f"df = pd.melt(df, id_vars={params.get('id_cols')}, "
        f"value_vars={params.get('value_cols')}, "
        f"var_name='{params.get('name_col')}', value_name='{params.get('value_col')}'")
    ),
    "Make wider": (
        f"df = df.pivot_table(index={params.get('index_cols')}, "
        f"columns='{params.get('column_source')}', "
        f"values='{params.get('value_source')}', "
        f"aggfunc={repr(params.get('aggfunc'))})"
    ),
    "Transpose": "df = df.transpose()",
    "Unmerge fill": "df = fill_merged_ranges(workbook, sheet_name, header_row)",
    "Rename columns": "df = df.rename(columns={...})",
}
return code_map.get(name, "# Code preview not available for this step")

```

def main() -> None:

```

    st.set_page_config(page_title="Reshape Wizard", layout="wide")
    init_state()

    st.title("Reshape Wizard (MVP)")
    st.caption("Offline-friendly reshaping tool with deterministic transformations and preview-first workflow.")

    # --- Sidebar: Import ---
    st.sidebar.header("Import")
    uploaded = st.sidebar.file_uploader("Upload CSV or XLSX", type=["csv", "xlsx", "xlsm", "xls"])
    selected_sheet = None
    header_row = st.sidebar.number_input("Header row (1 = first)", min_value=1, max_value=20,
value=1, step=1)
    unmerge_opt = st.sidebar.checkbox("Unmerge cells and fill values (preview/apply in Preprocess
tab)", value=False)
    ask_before_fix = st.sidebar.checkbox("Ask before fixing column names", value=False)

    file_bytes: Optional[bytes] = None
    if uploaded is not None:
        file_bytes = uploaded.getvalue()
        ext = uploaded.name.lower().split(".")[1]

```

```

if ext in ("xlsx", "xlsm", "xls"):
    try:
        sheets = get_sheet_names(file_bytes)
        selected_sheet = st.sidebar.selectbox("Sheet", sheets)
    except Exception as exc:
        st.sidebar.error(f"Could not read sheets: {exc}")

if st.sidebar.button("Load file", use_container_width=True):
    try:
        df, merge_info = load_dataframe(
            file_bytes=file_bytes,
            filename=uploaded.name,
            sheet_name=selected_sheet,
            header_row=header_row,
            unmerge_fill=False, # unmerge handled in Preprocess tab
        )
        st.session_state.upload_cache = {
            "bytes": file_bytes,
            "name": uploaded.name,
            "ext": ext,
            "sheet": selected_sheet,
            "header_row": header_row,
        }
        safe_cols, fixed = sanitise_columns(list(df.columns))
        if ask_before_fix and fixed:
            with st.sidebar.form("confirm_columns_load"):
                st.write("Some column names were empty or duplicated. Edit before applying.")
                user_cols = [
                    st.text_input(f"orig or '(empty)'", value=safe, key=f"colfix_load_{i}")
                    for i, (orig, safe) in enumerate(zip(df.columns, safe_cols))
                ]
                submitted = st.form_submit_button("Apply column names")
            if submitted:
                trimmed = [c.strip() for c in user_cols]
                if any(not c for c in trimmed):
                    st.sidebar.error("Column names cannot be empty.")
                    raise ValueError("Column names confirmation failed.")
                if len(trimmed) != len(set(trimmed)):
                    st.sidebar.error("Column names must be unique.")
                    raise ValueError("Column names confirmation failed.")
                safe_cols = trimmed
            else:
                st.sidebar.info("Auto-fixing column names. Submit above to override.")
        df.columns = safe_cols
        if fixed:
            st.sidebar.info("Some columns had empty or duplicate names and were auto-renamed.")
            st.session_state.original_df = df.copy()
            st.session_state.current_df = df.copy()
            st.session_state.history = []
            add_history_step(

```

```

        "Loaded",
        {
            "source": ext,
            "header_row": header_row,
            "sheet": selected_sheet,
            "unmerge_on_load": False,
            "columns_fixed": fixed,
        },
        df,
        before_shape=df.shape,
    )
    st.session_state.last_merge_fill = merge_info
    clear_preview()
    if df.shape[0] > LARGE_ROW_THRESHOLD:
        st.sidebar.warning(
            f"Large dataset ({df.shape[0]} rows). Preview limited to {preview_rows(df)} rows."
        )
    if unmerge_opt and ext in ("xlsx", "xlsm", "xls"):
        st.sidebar.info("Unmerge is ready in Preprocess tab. Preview before applying.")
    except Exception as exc:
        st.sidebar.error(f"Load failed: {exc}")

st.sidebar.markdown("---")
st.sidebar.subheader("Paste table from Excel")
pasted_text = st.sidebar.text_area("Paste your table here (Ctrl+V)", height=150)
if st.sidebar.button("Load pasted table", use_container_width=True):
    try:
        df = load_from_clipboard(pasted_text)
        if df.shape[1] == 1:
            st.sidebar.warning("Only one column detected; check that the table is tab-separated.")
        if df.shape[0] > LARGE_ROW_THRESHOLD:
            st.sidebar.warning(f"Pasted table has {df.shape[0]} rows; previews will be limited.")
        safe_cols, fixed = sanitise_columns(list(df.columns))
        if ask_before_fix and fixed:
            with st.sidebar.form("confirm_columns_clip"):
                st.write("Some column names were empty or duplicated. Edit before applying.")
                user_cols = [
                    st.text_input(f"orig or '(empty)'", value=safe, key=f"colfix_clip_{i}")
                    for i, (orig, safe) in enumerate(zip(df.columns, safe_cols))
                ]
                submitted = st.form_submit_button("Apply column names")
            if submitted:
                trimmed = [c.strip() for c in user_cols]
                if any(not c for c in trimmed):
                    st.sidebar.error("Column names cannot be empty.")
                    raise ValueError("Column names confirmation failed.")
                if len(trimmed) != len(set(trimmed)):
                    st.sidebar.error("Column names must be unique.")
                    raise ValueError("Column names confirmation failed.")
                safe_cols = trimmed
            else:

```

```

        st.sidebar.info("Auto-fixing column names. Submit above to override.")
df.columns = safe_cols
if fixed:
    st.sidebar.info("Some columns had empty or duplicate names and were auto-renamed.")
st.session_state.upload_cache = {}
st.session_state.original_df = df.copy()
st.session_state.current_df = df.copy()
st.session_state.history = []
add_history_step(
    "Loaded from clipboard",
    {"rows": df.shape[0], "cols": df.shape[1], "columns_fixed": fixed},
    df,
    before_shape=df.shape,
)
clear_preview()
st.sidebar.success("Pasted table loaded.")
except Exception as exc:
    st.sidebar.error(str(exc))

# --- Sidebar: Export & History ---
export_buttons(st.session_state.current_df)
render_history_sidebar()

current_df = st.session_state.current_df
df_hash = df_fingerprint(current_df) if current_df is not None else ""
auto_allowed = current_df is not None and current_df.shape[0] <= LARGE_ROW_THRESHOLD

# --- Main area ---
tabs = st.tabs(["Preview", "Preprocess", "Make longer", "Make wider", "Transpose", "Reproducible
code"])

with tabs[0]:
    show_before_after(current_df, current_df, {"after_shape": current_df.shape if current_df is not
None else (0, 0)}, None, "Current data")

# --- Preprocess tab (fill-down + unmerge option) ---
with tabs[1]:
    st.subheader("Preprocess")
    if current_df is None:
        st.info("Load data first.")
    else:
        fill_cols = st.multiselect("Columns to fill down", current_df.columns.tolist(), key="fill_cols")
        treat_empty = st.checkbox("Treat empty strings as missing", value=True, key="treat_empty")
        auto_update = st.checkbox(
            "Auto-update preview on parameter change",
            value=auto_allowed,
            key="auto_fill",
            disabled=not auto_allowed,
        )
        params = {
            "columns": fill_cols,

```

```

        "treat_empty": treat_empty,
    }
    params_hash = f"fill_down|{df_hash}|{json.dumps(params, sort_keys=True)}"
    trigger_preview(current_df, "fill_down", params, auto_update, params_hash, df_hash)
    preview_df, preview_meta, preview_err = get_preview_state(params_hash)
    apply_disabled = (
        preview_err is not None
        or preview_df is None
    )
    if st.button("Apply fill-down", key="apply_fill", disabled=apply_disabled):
        try:
            result_df, meta = _compute_preview(current_df, "fill_down", params)
            before_shape = current_df.shape
            st.session_state.current_df = result_df
            add_history_step(
                "Fill down",
                {**params, "warnings": meta.get("warnings", []), "diagnostics":
meta.get("diagnostics", {})},
                result_df,
                before_shape,
            )
            clear_preview()
            st.success("Fill-down applied.")
        except Exception as exc:
            st.error(f"Fill-down failed: {exc}")

    show_before_after(current_df, preview_df, preview_meta, preview_err, "After (preview)")

    st.markdown("---")
    st.subheader("Unmerge cells and fill (XLSX only)")
    cache = st.session_state.upload_cache
    if cache.get("ext") not in ("xlsx", "xlsm", "xls"):
        st.caption("Load an Excel file to enable unmerge preview.")
    else:
        sheet_choice = st.selectbox("Sheet for unmerge", cache.get("sheet"),
key="sheet_unmerge")
        header_choice = st.number_input(
            "Header row (1 = first)", min_value=1, max_value=20, value=cache.get("header_row", 1),
step=1, key="header_unmerge"
        )
        unmerge_params = {
            "cache_key": cache.get("name"),
            "sheet_name": sheet_choice,
            "header_row": int(header_choice),
        }
        unmerge_hash = f"unmerge|{json.dumps(unmerge_params, sort_keys=True)}"
        trigger_preview(current_df, "unmerge_fill", unmerge_params, auto_update,
unmerge_hash, df_hash)
        preview_df, preview_meta, preview_err = get_preview_state(unmerge_hash)
        apply_disabled_unmerge = (
            preview_err is not None

```

```

        or preview_df is None
    )
    if st.button("Apply unmerge+fill", key="apply_unmerge",
disabled=apply_disabled_unmerge):
        try:
            result_df, meta = _compute_preview(current_df, "unmerge_fill", unmerge_params)
            before_shape = current_df.shape
            st.session_state.current_df = result_df
            add_history_step(
                "Unmerge fill",
                {**unmerge_params, "warnings": meta.get("warnings", []), "diagnostics":
meta.get("diagnostics", {})},
                result_df,
                before_shape,
            )
            clear_preview()
            st.success("Unmerge + fill applied.")
        except Exception as exc:
            st.error(f"Unmerge failed: {exc}")
        show_before_after(current_df, preview_df, preview_meta, preview_err, "After (preview)")

st.markdown("---")
st.subheader("Rename columns")
with st.form("rename_columns_form"):
    new_names = [
        st.text_input(f"{{col}}", value=col, key=f"rename_{i}")
        for i, col in enumerate(current_df.columns)
    ]
    submitted = st.form_submit_button("Apply renames")
if submitted:
    trimmed = [c.strip() for c in new_names]
    if any(not c for c in trimmed):
        st.error("Column names cannot be empty.")
    elif len(trimmed) != len(set(trimmed)):
        st.error("Column names must be unique.")
    else:
        before_shape = current_df.shape
        renamed_df = current_df.copy()
        renamed_df.columns = trimmed
        st.session_state.current_df = renamed_df
        add_history_step(
            "Rename columns",
            {"mapping": dict(zip(current_df.columns, trimmed))},
            renamed_df,
            before_shape,
        )
        clear_preview()
        st.success("Columns renamed.")

# --- Make longer ---
with tabs[2]:

```

```

st.subheader("Make longer (melt)")
df = st.session_state.current_df
if df is None:
    st.info("Load data first.")
else:
    columns = df.columns.tolist()
    id_cols = st.multiselect("ID columns (keep)", columns, key="id_cols_longer")
    value_default = [c for c in columns if c not in id_cols]
    value_cols = st.multiselect("Columns to stack (values)", columns, default=value_default,
key="val_cols_longer")
    name_col = st.text_input("Name column", value="variable")
    value_col = st.text_input("Value column", value="value")

    auto_update = st.checkbox(
        "Auto-update preview on parameter change", value=auto_allowed, key="auto_longer",
disabled=not auto_allowed
    )
    params = {
        "id_cols": id_cols,
        "value_cols": value_cols,
        "name_col": name_col,
        "value_col": value_col,
    }
    params_hash = f"longer|{df_hash}|{json.dumps(params, sort_keys=True)}"
    trigger_preview(df, "longer", params, auto_update, params_hash, df_hash)
    preview_df, preview_meta, preview_err = get_preview_state(params_hash)

    apply_disabled = (
        preview_err is not None
        or preview_df is None
    )
    if st.button("Apply longer", key="apply_longer", disabled=apply_disabled):
        try:
            result_df, meta = _compute_preview(df, "longer", params)
            before_shape = df.shape
            st.session_state.current_df = result_df
            add_history_step(
                "Make longer",
                {**params, "warnings": meta.get("warnings", []), "diagnostics":
meta.get("diagnostics", {})},
                result_df,
                before_shape,
            )
            clear_preview()
            st.success("Longer applied.")
        except Exception as exc:
            st.error(f"Apply failed: {exc}")

    show_before_after(df, preview_df, preview_meta, preview_err, "After (preview)")

# --- Make wider ---

```



```

with tabs[3]:
    st.subheader("Make wider (pivot)")
    df = st.session_state.current_df
    if df is None:
        st.info("Load data first.")
    else:
        columns = df.columns.tolist()
        index_cols = st.multiselect("Index columns (rows)", columns, key="index_cols_wider")
        column_source = st.selectbox("Column names from", columns, key="col_source_wider")
        value_source = st.selectbox("Values from", columns, key="val_source_wider")
        agg_options = ["first", "mean", "sum", "count", "min", "max"]

        duplicates = df.duplicated(subset=index_cols + [column_source]).any() if column_source else
False
        if duplicates:
            st.warning("Duplicates detected in index/column combinations. Please choose an
aggregation.")
            aggfunc = st.selectbox("Aggregation (required if duplicates)", ["" ] + agg_options, index=0 if
not duplicates else 1, key="agg_wider")
            aggfunc = aggfunc or None
            reset_idx = st.checkbox("Reset index after pivot", value=True, key="reset_idx_wider")

            auto_update = st.checkbox(
                "Auto-update preview on parameter change", value=auto_allowed, key="auto_wider",
disabled=not auto_allowed
            )
            params = {
                "index_cols": index_cols,
                "column_source": column_source,
                "value_source": value_source,
                "aggfunc": aggfunc,
                "reset_index": reset_idx,
            }
            params_hash = f"wider|{df_hash}|{json.dumps(params, sort_keys=True)}"
            trigger_preview(df, "wider", params, auto_update, params_hash, df_hash)
            preview_df, preview_meta, preview_err = get_preview_state(params_hash)

            apply_disabled = (
                preview_err is not None
                or preview_df is None
            )
            if st.button("Apply wider", key="apply_wider", disabled=apply_disabled):
                try:
                    result_df, meta = _compute_preview(df, "wider", params)
                    before_shape = df.shape
                    st.session_state.current_df = result_df
                    add_history_step(
                        "Make wider",
                        {**params, "warnings": meta.get("warnings", []), "diagnostics":
meta.get("diagnostics", {})},
                        result_df,

```

```

        before_shape,
    )
    clear_preview()
    st.success("Wider applied.")
except Exception as exc:
    st.error(f"Apply failed: {exc}")

show_before_after(df, preview_df, preview_meta, preview_err, "After (preview)")
if index_cols and column_source:
    st.caption(f"Key uniqueness check: {'duplicates found' if duplicates else 'all unique'}")

# --- Transpose ---
with tabs[4]:
    st.subheader("Transpose")
    df = st.session_state.current_df
    if df is None:
        st.info("Load data first.")
    else:
        first_row_header = st.checkbox("First row becomes headers after transpose", value=True,
key="first_row_header")
        first_col_index = st.checkbox("First column becomes row names before transpose",
value=False, key="first_col_index")

        auto_update = st.checkbox(
            "Auto-update preview on parameter change", value=auto_allowed, key="auto_transpose",
disabled=not auto_allowed
        )
        params = {"first_row_header": first_row_header, "first_col_index": first_col_index}
        params_hash = f"transpose|{df_hash}|{json.dumps(params, sort_keys=True)}"
        trigger_preview(df, "transpose", params, auto_update, params_hash, df_hash)
        preview_df, preview_meta, preview_err = get_preview_state(params_hash)

    apply_disabled = (
        preview_err is not None
        or preview_df is None
    )
    if st.button("Apply transpose", key="apply_transpose", disabled=apply_disabled):
        try:
            result_df, meta = _compute_preview(df, "transpose", params)
            before_shape = df.shape
            st.session_state.current_df = result_df
            add_history_step(
                "Transpose",
                {**params, "warnings": meta.get("warnings", []), "diagnostics":
meta.get("diagnostics", {})},
                result_df,
                before_shape,
            )
            clear_preview()
            st.success("Transpose applied.")
        except Exception as exc:

```

```

st.error(f"Apply failed: {exc}")

show_before_after(df, preview_df, preview_meta, preview_err, "After (preview)")

with tabs[5]:
    st.subheader("Reproducible code")
    st.code(code_snippet_for_last_step(), language="python")

if __name__ == "__main__":
    main()

```

Great, thanks for pasting the code — that helps a lot.

Here are **Codex-ready tasks** to fix the three broken behaviours:

1. "Ask before fixing column names" not working
2. **Unnamed: 1** and empty names not handled consistently
3. Rename columns UI not reliably fixing things

Copy-paste this into Codex:

```

text

BUGFIX FEATURE: Robust handling of empty / unnamed / duplicate column names

PROBLEM
- Columns coming from Excel appear as "Unnamed: 1", "Unnamed: 2", or empty strings.
- "Ask before fixing column names" does not reliably stop auto-fixing.
- Rename columns form does not always update selectors or state correctly.
- Pasted tables may produce truly empty column names.

GOAL
Guarantee that:
1) All columns always have safe, usable names.
2) User can confirm or edit names before auto-fix.
3) Rename UI always works and updates app state.

-----
TASK 1 – Treat "Unnamed: X" as invalid column names
-----
Update is_invalid_colname(name):

Current:
- only checks NaN or ""

Change rule:
- name is invalid if:
  - NaN
  - empty or whitespace
  - matches regex: ^Unnamed(:\s*\d+)?$

So:
"Unnamed"
"Unnamed: 1"
"Unnamed: 2"
are treated as invalid and must be replaced.

-----

```

TASK 2 – sanitise_columns must preserve original names for user confirmation

Modify sanitise_columns to return:
safe_cols, fixed, mapping

Where mapping is:
list of (original_name, suggested_name)

So UI can show:
Old name → New name

Do NOT overwrite df.columns before user confirmation when:
ask_before_fix == True and fixed == True

TASK 3 – Fix "Ask before fixing column names" logic

Current behaviour auto-fixes even when ask_before_fix=True.

Change flow when loading (file or clipboard):

IF fixed AND ask_before_fix:
- show confirmation form
- DO NOT assign df.columns yet
- only assign df.columns = user-edited names after submit
- if not submitted → STOP load (return / do not continue pipeline)

No fallback auto-fix when ask_before_fix=True.

TASK 4 – Ensure clipboard import also goes through same column logic

After load_from_clipboard:
- ALWAYS call sanitise_columns
- Apply same confirmation logic as file load
- Do NOT allow df with empty or "Unnamed" columns to pass through

TASK 5 – Fix Rename Columns feature

Current bug: rename sometimes does nothing or breaks selectors.

Change Rename Columns to:

- Build mapping:
old_name → new_name
- Apply via:
df = df.rename(columns=mapping)

Then:
- Update:
st.session_state.current_df
st.session_state.history
- Call clear_preview()
- Trigger rerun with st.rerun()

Add validation:
- no empty names
- no duplicates
- strip whitespace

TASK 6 – Enforce column validity globally

After ANY operation that creates a new df (preview or apply):

- Run:
safe_cols, fixed, _ = sanitise_columns(df.columns)
- Assign:
df.columns = safe_cols
- If fixed:
add warning: "Some column names were auto-corrected."

Apply this to:
- unmerge_fill
- transpose

```
- pivot wider
- melt
```

TASK 7 – Acceptance tests

Must pass:

- 1) Excel file with blank header cells:
→ columns become Column_1, Column_2, etc.
- 2) Excel with Unnamed: 1:
→ treated as invalid and renamed.
- 3) ask_before_fix=True:
→ user must confirm names before data loads.
- 4) Rename columns:
→ immediately reflected in all selectors.
- 5) Clipboard paste with empty columns:
→ columns auto-generated or user-confirmed.
- 6) No operation crashes due to bad column names.

SUCCESS CRITERIA

- There is never a column with name:
"", NaN, or "Unnamed:*"
- User always has a chance to fix names when requested.
- Rename feature always updates the dataframe and UI.

If you want, next I can:

- ✓ rewrite your `sanitise_columns()` + `is_invalid_colname()` properly
- ✓ or rewrite the load logic block correctly
- ✓ or produce a **patch-style diff** for this file

Just say which you want:

functions, **load logic**, or **full patch**.



is it possible to make preview like SPSS like being able to move columns and rows as USER in the table preview, short answer

Partly.

In Streamlit's built-in `st.dataframe`, you **can't** let users manually drag columns/rows around like in SPSS.

But you can get close by:

- adding a **column order control** (multiselect / up-down buttons) that reorders the preview, and
- using a grid component like **streamlit-aggrid** to allow **interactive column dragging/sorting/filtering** in the preview.

True "freeform row reordering by dragging" is much harder (and usually not worth it), but **column drag + sort/filter** is doable.



i guess i would like to have one page where the table where we would have "list of variables" and drag them into rows and columns
also is it possible to add option copy table, instead of downloading it?
short answers pls

Yes to both.

1. Drag variables into Rows / Columns (SPSS-style pivot UI)

Yes, it's possible.

You'd need a custom grid component (e.g. `streamlit-aggrid` or a custom React component) to support **drag & drop**.

With pure Streamlit widgets → only dropdowns/multiselects, not true drag.

2. Copy table instead of downloading

Yes, easy.

You can add a **"Copy to clipboard"** button that copies the table as:

- TSV (best for Excel paste), or
- CSV.

So:

- ✓ SPSS-like drag layout = possible (extra component)
- ✓ Copy instead of download = easy and recommended



let's do both
create a task pls for COdex

Here is a **short, Codex-ready task** for both features:

text

FEATURE 1: SPSS-like drag variables into Rows / Columns

USER STORY

As a user, I want to drag variables into "Rows" and "Columns" areas (like SPSS), so I can define pivot structure visually instead of using dropdowns.

IMPLEMENTATION

- Use `streamlit-aggrid` (or similar grid component with drag support).
- Layout:
 - Left panel: list of variables (`df.columns`)
 - Right panel:
 - Drop zone: Rows
 - Drop zone: Columns
 - Optional: Values (for pivot)
- User drags column names into:
 - Rows → `index_cols`
 - Columns → `column_source`
 - Values → `value_source`
- Map drag result to existing Make Wider (pivot) logic.

- Fallback:
If drag UI not available → use existing multiselects.

ACCEPTANCE

- User can drag columns into Rows and Columns.
- Pivot preview updates based on drag result.
- Dragged layout persists in session_state.

FEATURE 2: Copy table to clipboard (instead of download)

USER STORY

As a user, I want to copy the current or preview table to clipboard, so I can paste it directly into Excel or Google Sheets.

IMPLEMENTATION

- Add button: "Copy table to clipboard"
- On click:
 - Convert DataFrame to TSV:
df.to_csv(sep="\t", index=False)
 - Use:
st.code(tsv_text) + JavaScript clipboard API
OR streamlit.components.v1.html with:
navigator.clipboard.writeText(...)
- Support:
 - copy current_df
 - copy preview_df (if available)
- Show success message: "Table copied to clipboard."

EDGE CASES

- Limit copy size (e.g. max 200k cells); warn if exceeded.
- Strip index unless user explicitly includes it.

ACCEPTANCE

- 1) Click "Copy table" → paste into Excel → table appears correctly in cells.
- 2) Works for both preview and applied result.
- 3) No file download needed.

If you want, next I can:

- ✓ write the exact Streamlit + aggrid code
- ✓ or integrate this into your existing pivot tab
- ✓ or give a minimal example just for drag UI

Just say which.



okay write task:

1. let's remove drag and drop cos it doesn't work
shows constantly "Drag UI unavailable; falling back to selectors."
2. Need to have copy table on each page
3. on preview page current data isn't updating even when Apply is clicked on other pages, it appears only when other action on preview page is done, like download xlsx, so we need to fix it either add update button on Preview page, or make sure it updates after Apply done on other pages

Here is a **Codex-ready task** covering all 3 fixes, short and precise:

text

BUGFIX TASKS: Remove drag UI, add copy table, fix Preview refresh

TASK 1 – Remove drag-and-drop UI for Rows/Columns

PROBLEM:

Drag UI constantly shows “Drag UI unavailable; falling back to selectors.”

SOLUTION:

- Completely remove drag-and-drop variable UI.
- Remove:
 - streamlit-aggrid (if only used for drag)
 - any code that renders drag areas or shows “Drag UI unavailable”.
- Use only:
 - multiselect / selectbox controls for pivot (Rows, Columns, Values).
- Ensure Make Wider UI uses only selectors.

ACCEPTANCE:

- No drag-related messages appear.
- UI only shows dropdown/multiselects.

TASK 2 – Add “Copy table” button on every page

USER STORY:

User wants to copy table instead of downloading.

IMPLEMENTATION:

- Add button: “Copy table to clipboard”
- Present on:
 - Preview tab
 - Preprocess tab
 - Make longer tab
 - Make wider tab
 - Transpose tab
- Behaviour:
 - If preview_df exists → copy preview_df
 - Else → copy current_df
- Copy format:
 - df.to_csv(sep="\t", index=False)
- Use clipboard API via:
 - streamlit.components.v1.html with JS:
 - navigator.clipboard.writeText(text)

EDGE:

- If table too large (e.g. >200k cells) → warn and block.

ACCEPTANCE:

- Click “Copy table” → paste into Excel → correct table appears.
- Works for preview and applied data.

TASK 3 – Fix Preview tab not updating after Apply

PROBLEM:

Preview tab shows stale data until user clicks something there.

CAUSE:

Streamlit does not rerender Preview tab after Apply on other tabs.

SOLUTION:

After every successful Apply action:

- Call:
 - st.rerun()

OR

- Add explicit “Refresh Preview” button on Preview tab that:
 - sets:
 - st.session_state.preview_df = None
 - st.session_state.preview_meta = None
 - st.session_state.preview_error = None

REQUIRED CHANGE:

After:

```
Apply fill-down
Apply unmerge
Apply longer
Apply wider
Apply transpose
Apply rename
```

Add:

```
st.rerun()
```

ACCEPTANCE:

- 1) Apply operation in any tab → Preview tab shows updated current_df immediately.
- 2) No need to click Download or any other control to refresh.
- 3) Preview always reflects st.session_state.current_df.

If you want, I can next:

- ✓ write exact code snippets for **copy to clipboard**
- ✓ or give you a patch for the Preview refresh bug
- ✓ or merge this into your master Codex spec

Just say which.



1. is it possible also to make tables wider by combining values in columns? like in table

Needs_aggregation

Person	Year	ScoreType	Score
Alice	2023	Math	80
Alice	2023	English	90
Bob	2023	Math	70
Bob	2023	English	75

i would like to be able to create output like

```
Person Math_2023 English_2023
Alice 80 90
```

add customisable separators

?

2. If 1 is yes, can we do opposite and use customisable separators to separate columns and reverse what we did in previous?

short answer

1. Yes.

You can widen the table and **combine multiple fields into column names** (e.g. Math_2023, English_2023) with a **custom separator** (_, |, -, etc.).

This is just pivoting with:

- columns = ScoreType + separator + Year.

2. Yes (mostly).

You can reverse it by:

- **splitting column names by the same separator** into multiple columns (e.g. `Math_2023` → `ScoreType=Math, Year=2023`),
- then pivoting longer.

Condition:

- ✓ works reliably if the separator is unique and consistent
- ⚠️ ambiguous if values themselves contain the separator

So:

Yes + Yes (with a clean separator rule).



let's write a task for codex

Here is a **short, Codex-ready task** for this feature:

text

FEATURE: Pivot wider with combined column names + reverse (split back)

PART 1 – Combine multiple columns into new wide column names

USER STORY

As a user, when pivoting wider, I want to build new column names by combining several fields (e.g. `ScoreType + Year`), using a custom separator (e.g. `"_"`, `"-"`, `"|"`).

EXAMPLE

Input:

Person, Year, ScoreType, Score
Alice, 2023, Math, 80
Alice, 2023, English, 90

Settings:

- Index cols: Person
- Column name parts: [ScoreType, Year]
- Separator: `"_"`
- Values from: Score

Output:

Person, Math_2023, English_2023
Alice, 80, 90

IMPLEMENTATION

- UI:
 - Multiselect: "Column name parts" (1+ columns)
 - Text input: "Separator" (default `"_"`)
- Logic:
 - Create temp column:


```
df["_combined_col"] = df[col1].astype(str) + sep + df[col2].astype(str) (+ more if selected)
```
 - Use pivot:


```
pivot(index=index_cols, columns="_combined_col", values=value_source)
```
- Apply duplicate detection + aggregation same as normal pivot wider.

PART 2 – Reverse: Split combined column names back to long form

USER STORY

As a user, I want to split wide columns like `"Math_2023"`

back into separate variables (ScoreType, Year) and unpivot.

IMPLEMENTATION

- UI:
 - Multiselect: "Columns to split" (e.g. Math_2023, English_2023)
 - Text input: "Separator"
 - Text inputs: "New column 1 name", "New column 2 name" (e.g. ScoreType, Year)
 - Value column name (e.g. Score)
- Logic:
 - Melt selected wide columns into long:
 - var_name = "combined"
 - value_name = value_col
 - Split "combined" by separator into multiple new columns.
- Validation:
 - Separator must exist in all selected column names.
 - Block if inconsistent split lengths.

ACCEPTANCE TESTS

- 1) Combine ScoreType + Year → Math_2023 works.
- 2) Custom separator "-" → Math-2023 works.
- 3) Reverse split of Math_2023 → ScoreType=Math, Year=2023 works.
- 4) Warn if separator not found or ambiguous.

If you want, I can next:

- ✓ integrate this into your existing "Make wider" and "Make longer" tabs
- ✓ or write the actual pandas code blocks
- ✓ or merge this into your master Codex task list

Just say which.



what this might mean

```
KeyError: Index(['_combined_wider'], dtype='object')
```

Traceback:

```
File "/Users/pavlosereda/Test/app.py", line 1152, in <module>
    main()
```

```
File "/Users/pavlosereda/Test/app.py", line 1000, in main
```

```
    duplicates = df.duplicated(subset=index_cols + [column_used]).any() if column_used else False
```

```
File "/Users/pavlosereda/anaconda3/envs/py3.10/lib/python3.10/site-
packages/pandas/core/frame.py", line 6698, in duplicated
```

```
    raise KeyError(Index(diff))
```

Short meaning:

👉 **Pandas is trying to use a column that does not exist in the DataFrame.**

Specifically:

'_combined_wider' is **not in df.columns** at the moment this line runs:

```
python
```

```
df.duplicated(subset=index_cols + [column_used])
```

So one of these is true:

1. You **expected** `_combined_wider` to exist, but:

- it was never created, or
- it was created on a different dataframe (e.g. `df_temp`) and you're now using `df`.

2. `column_used` is set to `"_combined_wider"` but:

- `df.columns` does NOT contain it (check with `print(df.columns)`).

3. You created it conditionally (only in some cases), but your duplicate-check runs **before** that creation.

Typical cause in your case:

You are doing something like:

```
python

df["_combined_wider"] = ...
```

but later you call:

```
python

duplicates = df.duplicated(subset=index_cols + [column_used])
```

where `column_used = "_combined_wider"`, **but the df here is the original one**, not the modified one.

So:

✓ Error = "you referenced a column that doesn't exist"

🔧 Fix = ensure `_combined_wider` is created **before** any logic that uses it, and on the **same df variable**.



```
it gives it when i try for pivot wider multiple columns or any columns in Column name parts
(optional, to combine)
KeyError: Index(['_combined_wider'], dtype='object')
Traceback:
File "/Users/pavlosereda/Test/app.py", line 1152, in <module>
    main()
File "/Users/pavlosereda/Test/app.py", line 1000, in main
    duplicates = df.duplicated(subset=index_cols + [column_used]).any() if column_used else False
File "/Users/pavlosereda/anaconda3/envs/py3.10/lib/python3.10/site-
packages/pandas/core/frame.py", line 6698, in duplicated
    raise KeyError(Index(diff))
```

That error means:

👉 your code is **checking duplicates on** `_combined_wider` **before that column actually exists in df**.

When you select "Column name parts (to combine)", your logic probably does:

- sets `column_used = "_combined_wider"`

- BUT the line that *creates* it (something like
`df["_combined_wider"] = df[col1] + "_" + df[col2]`
 runs **after** this line:

```
python

duplicates = df.duplicated(subset=index_cols + [column_used])
```

So pandas says:

✗ `_combined_wider` is not in `df.columns`

In short:

You reference `_combined_wider` too early.

Fix conceptually:

- Create `_combined_wider` first
- THEN run `duplicated(...)` and pivot on it
- And make sure it's created on the same `df` object you pass to pivot

Typical correct order:

```
python

if combine_cols:
    df = df.copy()
    df["_combined_wider"] = df[combine_cols].astype(str).agg(sep.join, axis=1)
    column_used = "_combined_wider"
else:
    column_used = column_source

duplicates = df.duplicated(subset=index_cols + [column_used]).any()
```

Right now your order is reversed, so pandas can't find the column.



write a task for codex

Here is a **Codex-ready task** to fix this exact error:

```
text

BUGFIX: KeyError when using "Column name parts (to combine)" in pivot wider

PROBLEM
When user selects one or more "Column name parts (optional, to combine)",
the app crashes with:
KeyError: '_combined_wider'

Cause:
- Code checks duplicates using:
  df.duplicated(subset=index_cols + [column_used])
- But when column_used == "_combined_wider",
  that column has NOT been created in df yet.
```

GOAL

Always create the combined column BEFORE:

- duplicate detection
- pivot
- any reference to "_combined_wider"

TASK 1 – Create combined column first

In Make Wider logic:

Current (wrong order):

- decide column_used = "_combined_wider"
- run duplicated(...)
- later create df["_combined_wider"]

Change to:

IF combine_columns is not empty:

```
df_work = df.copy()
df_work["_combined_wider"] =
    df_work[combine_columns].astype(str).agg(separator.join, axis=1)
column_used = "_combined_wider"
```

ELSE:

```
df_work = df
column_used = column_source
```

Then:

```
duplicates = df_work.duplicated(subset=index_cols + [column_used]).any()
```

And use df_work for:

- duplicate detection
- pivot / pivot_table

TASK 2 – Never reference "_combined_wider" unless it exists

Before any use of column_used:

- Assert:


```
if column_used not in df_work.columns:
    raise ValueError("Combined column was not created correctly.")
```

TASK 3 – Preview and Apply must use same logic

Ensure both:

- preview (_compute_preview)
- apply (Apply wider button)

use the same combined-column creation code path,
not duplicated logic.

ACCEPTANCE TESTS

- 1) Select Column name parts = ["ScoreType", "Year"], separator "_"
 - no KeyError
 - duplicates check works
 - output columns = Math_2023, English_2023
- 2) Select only Column names from (no combine)
 - behaviour unchanged
- 3) Duplicates present → aggregation is still required

If you want, I can also:

- ✓ write the exact pandas code block
- ✓ patch your _compute_preview("wider") section
- ✓ or produce a minimal diff for your file

Just say which.