Teddy

I need to build a desktop application that contains a full pipeline for OCR and TEI from images of text.

The primary use cases will consist of printed books with prose, poetry, and plays, as well as manuscript letters. That means that individual images may be provided in common formats like JPEG or in files with many images like PDF. They will generally contain a good deal of text, however the layout and formatting may vary a great deal to various ends. In some cases, the running text will simply be printed in two-column format. In other cases, headings will be spatially separated from the main text in ways that indicate their bibliographic role, apart from the text's main content.

Output from the program should include the main text, as well as markup with information about format and layout. In addition, the markup should include bibliographic information about the text itself. For example, the markup in output files should distinguish between the main body of a text and headers that run above it on each page, or among title page, table of contents, main body, and index. From the output file, users should be able to extract the main text of a book or letter, with all the words in the correct reading order, as well as bibliographic information like title, author, and publication or manuscript date. It will not be possible to identify that information in all cases. The program will need to be robust to those cases.

The markup should follow the specifications in the TEI P5 Guidelines, and output should be presented in XML format. The TEI P5 Guidelines are uploaded to this chat as a related file here. TEI is highly customizable. The same schema should be used for all documents. The schema should also be relatively simple. It is not necessary to create TEI files as an intermediate step when processing new texts. Markup may occur directly in XML.

Several examples of TEI markup from EBBA and ECCO are uploaded in the project files. There are two versions of each text: an image file (PDF, JPEG, PNG) containing scans of the documents and XML files containing the marked up text. The image file and markup file share filenames that differ only by filetype extension. In addition to the examples in the project files, browse to the following URLs for TEI snippets and discussion of best practices for prose books, poetry, drama, and primary (handwritten) sources.

Prose: https://www.teibyexample.org/exist/examples/TBED03v00.htm
Poetry: https://www.teibyexample.org/exist/examples/TBED04v00.htm
Drama: https://www.teibyexample.org/exist/examples/TBED05v00.htm
Primary: https://www.teibyexample.org/exist/examples/TBED06v00.htm

Note that some pages have images, which should be inspected visually for comparison to the TEI snippets.

Consider bibliographic and textual features that are shared among all examples (they may use different terms or rely on different hierarchies of objects). Consider also the features that are unique to each genre of writing (such as the speaker's name for lines of dialogue in drama or words that are struck out in manuscript). Develop a schema that incorporates them parsimoniously.

The desktop app should prioritize ease of installation, ease of use, and efficient processing during run time. Installing the app should not require the command line. The target user has no prior experience with the command line. The user should be able to run an installation management program or drag an icon into the Applications folder in order to install the program. Also, when the program has been installed, there should not be a separate folder of resources in the Applications folder -- all resources should be stored within the app.

The program should be written in Python. The language and any necessary libraries should be installed as part of the main program's installation process. The minimum requirements of the program will be a GPU with 16GB memory. That means that the program can use machine vision models through HuggingFace, such as Qwen3-VL. The implementation will have to be highly optimized. For example, pay attention to model weight precision and quantization, and try to use techniques like flash attention. Depending on the model's own capabilities, the program may (1) prompt the model for extracted text and apply TEI markup afterward, or (2) prompt the model to return text with markup applied. If the program does the latter, there must be TEI validation and possible revision before serving the XML file to the user.

For an overview of Open Source OCR models, browse to the URL:
https://huggingface.co/blog/ocr-open-models

For an overview of efficient inference with HuggingFace, see the following URLs:
https://huggingface.co/docs/diffusers/en/optimization/fp16
https://huggingface.co/docs/diffusers/en/optimization/cache
https://huggingface.co/docs/diffusers/en/optimization/attention_backends
https://huggingface.co/docs/diffusers/en/optimization/memory
https://huggingface.co/docs/diffusers/en/optimization/speed-memory-optims

The application should have a graphical user interface. It should be modern looking with a light background, like Google Drive, GitHub, or Claude. There should be brief instructions to the user where decisions need to be made or where information should be input. The app must allow users to upload image files. There should also be an opportunity to select any relevant parameters for the scan and to enter any bibliographic metadata that may be useful to include. If the process of OCR and text markup will be more than a few seconds, then there should be something like a progress bar to indicate what steps are being taken and/or how long to expect the process to take. After generating the XML file, there should be a window to previous the file, and a button to save the file to disk.

Claude

I'll start by examining the project files and reading the relevant skill files, then research the TEI examples and technical resources before designing the application.Now let me examine the TEI XML examples to understand the markup patterns:Now let me fetch the TEI By Example pages for best practices:Now let me check the OCR models blog post:Now let me check HuggingFace optimization docs briefly:Now I have a thorough understanding of the requirements. Let me also check the TEI Guidelines PDF briefly for the schema design:I now have comprehensive understanding of the requirements. Let me build the complete application. This is a large, complex project that I'll structure carefully.Now I'll create the full application. Let me start with the core Python modules:Now let me create the GUI:Now create the build/packaging configuration:Now create the comprehensive README:

[Interrupted due to context limit]

Hmm, let me fix that truncated file and write it completely:The file creation cut off at the backtick inside the code. Let me write the README with escaped content:Now let me create the build script for easy packaging:Now let me verify the complete project structure and make sure everything is wired up:Let me now run a basic syntax check to verify the Python code is valid:Let me also verify the TEI schema module independently since it's the most critical:Now let me test the TEI generator module with mock OCR output:Let me also test poetry and drama markup:The core TEI generation is working. Let me also test the manuscript markup:The manuscript output correctly generates `<opener>`, `<dateline>`, `<salute>`, `<del>`, `<add>`, `<unclear>`, `<gap>`, `<closer>`, and `<signed>` elements. I notice the closing/signed text appears both in the paragraph and in the closer — let me fix that ordering issue in the generator:Now copy the final project to the outputs:Here's an overview of the complete TEI-OCR application:

Architecture

The application has **three layers**:

1. Core Pipeline (`core/`)

- **`image_loader.py`** — Loads JPEG/PNG/TIFF images and extracts pages from multi-page PDFs (using PyMuPDF or pdf2image), with automatic resizing for VRAM management
- **`ocr_engine.py`** — Wraps Qwen2.5-VL (7B or 3B) with 4-bit NF4 quantization, Flash Attention 2, and bfloat16 precision to fit in 16GB VRAM. Prompts the model to return text with structural annotation markers (e.g., `[HEADING]`, `[ITALIC]`, `[SPEAKER]`, `[STANZA]`, `[DELETION]`), genre-tuned per document type

- `tei_generator.py` — Converts the annotated OCR text into a TEI P5 XML document tree using lxml, with genre-specific parsers for prose, poetry, drama, and manuscripts
- `tei_schema.py` — Defines the unified TEI P5 schema, creates skeleton documents, validates structure, and handles XML repair
- `pipeline.py` — Orchestrates the full workflow with progress tracking and cancellation support

2. GUI (`gui/`)

- `main_window.py` — PySide6 (Qt6) interface with drag-and-drop file upload, genre/model selectors, bibliographic metadata fields, progress bar, XML preview with monospace font, validation tab, and save dialog
- `styles.py` — Clean light theme stylesheet inspired by Google Drive/GitHub/Claude

3. Packaging (root)

- `build.spec` — PyInstaller config that creates a macOS .app bundle or Windows executable with all resources inside
- `build_installer.py` — Build automation script
- `setup.py` / `requirements.txt` — Standard Python packaging

TEI Schema Design

The schema is **unified across all genres** — a single set of elements covers prose, poetry, drama, and manuscripts. Key design decisions drawn from the EEBO/ECCO examples and TEI By Example best practices:

- `<fw>` for forme work (running headers, page numbers, catchwords, signatures) — keeping them separate from body text per the EEBO pattern
- `<hi @rend>` for typographic features (italic, bold, smallcaps, blackletter, decorative initials)
- `<lg>/<l>` for verse; `<sp>/<speaker>/<stage>` for drama; `<add>/<del>/<unclear>/<gap>` for manuscripts
- `<opener>/<closer>/<signed>/<dateline>` for letter structure
- `<front>/<body>/<back>` with typed `<div>` elements for document sections

A few aspects require revising:

First, the application does not have to be written in Python alone. Consider using other languages or frameworks to generate the interface. For example, consider running the program in the desktop machine's web browser.

Second, program must be capable of running on Apple Silicon. That means potentially using the 'mps' backend in PyTorch, rather than 'cuda'.

Third, the installation process requires the user to run the command line in order to install Python dependencies -- and it assumes that the user already has Python installed on their machine. If possible, provide a package installer that will automate the steps. Otherwise, find another way to streamline installation.

Let me re-examine the architecture with these constraints in mind. The key issues are: (1) the GUI should potentially be browser-based, (2) Apple Silicon/MPS support is needed, (3) installation must not require the command line or pre-installed Python.

Let me research the best approach for a self-contained desktop app with a web UI:Good. I now have a clear picture of the revised architecture. Let me now rebuild the application with these three critical changes:

1. **Electron + Flask** architecture: HTML/CSS/JS frontend in Electron, Python backend via Flask HTTP server
2. **MPS backend support** for Apple Silicon (with CUDA fallback for NVIDIA)
3. **Bundled Python** via python-build-standalone + PyInstaller, so no CLI installation needed

Let me clean up and rebuild:Let me build the new version with all the files. I'll start with the Python backend (Flask API), then the Electron frontend, then the build system.Now the OCR engine with MPS support:Now I'll copy the TEI generator and pipeline from the previous version (they don't change), then create the Flask server and the Electron frontend:Now the Flask API server:Now the Electron frontend:Now the HTML/CSS/JS frontend:Now the build script that packages everything together:Now let me verify everything:That's expected — torch isn't installed in this environment. The device module requires torch. Let me make the import lazy:Also need to fix the `get_torch_dtype` function:Let me verify the full project structure:

[Interrupted due to context limit]

It looks like you've re-sent the same three revision requests, but I've already addressed all three in the latest version of the project. Let me confirm what's already been done:

1. Web-based UI (not Python-only): The current version uses **Electron + HTML/CSS/JS** for the frontend, with a Python Flask backend. The UI runs in Chromium via Electron — it's a web interface in a desktop window.

2. Apple Silicon / MPS support: The `core/device.py` module detects MPS and sets `PYTORCH_ENABLE_MPS_FALLBACK=1`automatically. The `ocr_engine.py` loads models with `float16` on MPS (no quantization needed thanks to unified memory), `bfloat16` + 4-bit NF4 on CUDA, and `float32` on CPU.

> **3. No command-line installation:** The `build.py` script uses **PyInstaller** to bundle the entire Python runtime + all dependencies into a standalone binary. **electron-builder** then wraps that binary inside the Electron app as an `extraResource`. The result is a native installer (`.dmg` on macOS, `.exe` on Windows, `.AppImage` on Linux) — the user just double-clicks to install. No Python, no pip, no terminal.
>
> If there's a specific aspect of any of these three points you'd like me to refine further, let me know and I'll revise accordingly. Otherwise the project files are ready — shall I re-present them?

Yes, please re-present the files.

> Here are all 15 files. The README covers architecture, platform support, and installation for each approach.

A few revisions:

First, let's simplify the Web-based UI by removing Election and Node.js. Try to use Flask alone.

Second, while installing Python packages through pip, the version of PyTorch that was installed specified it was for CPU. Ensure that the appropriate GPU version gets installed when available.

Third, skip the build.py script. We don't need to create an installer after all. The dependencies are very machine specific. An installer would only be useful on my machine. Instead, simply enable me to create the app in my Applications folder. (And again, please do not create an additional folder inside of Applications with resources for the program. Include them within the app itself.)

> Let me rethink the architecture based on your feedback. The key changes are:
>
> 1. Drop Electron entirely — Flask serves the UI directly in the user's browser
> 2. Fix PyTorch installation to get GPU-accelerated versions (MPS for Apple Silicon, CUDA for NVIDIA)
> 3. Replace the installer with a simple macOS .app bundle that contains everything inside itself
>
> Let me research the macOS .app bundle structure and PyTorch installation:Good. Key insight: on macOS Apple Silicon, the **default** `pip install torch` already includes MPS support — the issue the user hit was likely using `--index-url https://download.pytorch.org/whl/cpu` which forces CPU-only wheels. The standard install from PyPI on macOS arm64 is already GPU-ready.
>
> Now I have a clear picture. Let me rebuild the project:Now the Flask server that serves the UI directly:Now the HTML template served by Flask:Now copy over the core modules (they're mostly the same as the previous revision):Now the README:Now verify

everything:Core modules all pass. The Flask import failure is expected since flask-cors isn't installed in this sandbox. Let me verify the file structure:Here are all 12 files. The key changes from the previous version:

Electron/Node.js removed. Flask serves the HTML UI directly at `http://127.0.0.1:5199`. The browser is the frontend.

PyTorch GPU installation fixed. The `setup_app.py` script detects your platform and installs PyTorch accordingly. On Apple Silicon, it uses the plain `pip install torch` (which includes MPS support by default) — it does *not* use `--index-url .../cpu`, which was likely the cause of the CPU-only build you encountered.

macOS .app bundle, nothing outside it. Running `python3 setup_app.py` creates `/Applications/TEI-OCR.app` with the venv, all dependencies, source code, and launcher all inside `Contents/Resources/`. To uninstall, just delete the .app.

Installation returned a warning:

```
WARNING:
/Users/edwinroland/Desktop/research/claude-code/tei-demo/tei-ocr/core
not found. Copy core/ manually.
```

And the program returns an Internal Server Error when the icon is double-clicked:

```
The server encountered an internal error and was unable to complete
your request. Either the server is overloaded or there is an error in
the application.
```

The setup script is looking for `core/` relative to `setup_app.py`'s location, but the file paths on your machine differ from what it expects. Let me also check what's actually happening inside the .app bundle and fix the launcher.

Let me trace through both issues:Found the issues:

1. **`Path(__file__).parent`** — if the user ran `python3 setup_app.py` from a different working directory, or if `__file__` resolves to a symlink, the `source_dir` might not be where `core/`, `server.py`, and `templates/` actually live. More importantly, the script only copies `server.py` but not `__main__.py`, and uses a fragile single-file loop.
2. **The launcher runs `python -m server`** — but the working directory and module path setup may be wrong. The server needs the `templates/` folder to be findable by Flask and `core/` to be importable.

Let me fix both:Now let me also fix the server.py — the Internal Server Error is likely because Flask can't find the `templates/` directory when running from inside the .app bundle. The launcher `cd`s into the app dir, but Flask's `template_folder`needs to be an absolute path or correct relative path:The issue: `Flask(__name__, template_folder="templates")` uses a path relative to the Python file's `__name__` module location, which may not match the working directory. When running from inside the .app bundle, the server.py is at `Resources/app/server.py` and templates is at `Resources/app/templates/`. The `cd "$APP_DIR"` in the launcher should make this work, but let me make it bulletproof by using an absolute path:Now verify everything:Two fixes in this update:

1. Source file copy failure — The old script used `Path(__file__).parent` without `.resolve()`, which could produce wrong paths depending on how you invoked it (different working directory, symlinks, etc.). The new version uses `Path(__file__).resolve().parent` and runs a pre-flight check that verifies `server.py`, `__main__.py`, `core/`, and `templates/` all exist at that location *before* doing anything else. If anything is missing, it prints a clear error and exits instead of silently skipping.

2. Internal Server Error — Flask was initialized with `template_folder="templates"` (a relative path), which resolves relative to the module's package location, not the working directory. When the launcher `cd`s into the app dir, these could disagree. The fix computes an absolute path from `server.py`'s own location: `Path(__file__).resolve().parent / "templates"`. This works identically whether you're running from the source tree or from inside the `.app` bundle.

I don't have the folders "core/" and "templates/". Present those to me.

The files are all there in the outputs — `core/` has 7 files and `templates/` has `index.html`. The issue is that when I present files, only individual files are downloadable, not folders. Let me re-present them grouped clearly so you can reconstruct the directory structure:

`core/` directory (7 files):**`templates/` directory** (1 file):Download all of these and place them so your project directory looks like:

tei-ocr-app/

├── setup_app.py

├── server.py

├── __main__.py

```
├── README.md
├── core/
│   ├── __init__.py
│   ├── device.py
│   ├── image_loader.py
│   ├── ocr_engine.py
│   ├── pipeline.py
│   ├── tei_generator.py
│   └── tei_schema.py
└── templates/
    └── index.html
```

The `core/` files go inside a folder named `core` and `index.html` goes inside a folder named `templates`, both at the same level as `setup_app.py`. Once they're in place, `python3 setup_app.py` should pass the pre-flight check and copy everything into the `.app`bundle.

The app works. While it runs, it hangs for a while at "Loading for Apple Silicon". What is happening in that step? Simply loading the model into memory? Or downloading the model?

Both things, potentially. Here's what happens in that step:

First run: The model weights (~8 GB for the 7B model) are **downloaded** from Hugging Face to a local cache, typically at `~/.cache/huggingface/hub/`. This is the slow part — it depends entirely on your internet speed and could take anywhere from a few minutes to 30+ minutes.

Every run (including the first): The weights are **loaded from disk into memory**. On Apple Silicon this means reading ~14 GB of float16 weights into unified RAM. Even from a fast SSD this takes a noticeable amount of time — typically 30–90 seconds depending on model size and your Mac's memory bandwidth.

The server gives no feedback to the browser during this step because the model loading is a blocking call inside `transformers`. I can improve this by adding progress

> messages that get sent back to the UI — would you like me to make the loading step report its status (downloading vs. loading, with progress percentage)?

The program runs successfully. Let's add a few features and do some minor debugging. Also, change the name of the application to "TEI Pipeline".

New features:

The model should try to infer bibliographic metadata (Title, Author, Date, Publisher, and Place) from the image. Rather than ask for that information before scanning, there should be a step to confirm the inferred metadata after the text has been OCR'ed and to allow the user to edit it. Inferred values should initially be included in the TEI markup, visible after processing. However, if the user edits the metadata, the TEI markup file will need to be updated (and visibly refreshed in the window).

Language should be optionally inferred also. In addition to the languages listed in the dropdown menu, there should be an autodetect button as well. The inferred language refers to the main text, which sometimes differs from that of the front matter.

Users should be able to extend the tagset before inference. They should have the option to suggest new elements to tag, such as <PERSON> or <PLACE> or <EXTENTOFOUVRE>. Users should provide the name of a tag and a short description of the text spans it should apply to. Consider that applying the tags to the text requires that they conform to proper XML and cannot violate the bounds of surrounding tags.

Loading the model for each image is time consuming. If possible, load the model on starting the program and keep it in memory while the user uploads new images into the program. Generate a simple, abstract icon for the application in keeping with the program's aesthetics. This should be the application's icon that the user clicks on.

Debugging:

The application icon appears in the Applications folder with a cancel sign over it. Double clicking the icon successfully runs the program, however the icon bounces in the dock much longer than usual. Try to resolve both issues.

A four page, 700k file caused a memory overflow in the GPU, with the following message:

"Error: MPS backend out of memory (MPS allocated: 24.66 GiB, other allocations: 8.53 MiB, max allowed: 30.19 GiB). Tried to allocate 5.61 GiB on private pool. Use PYTORCH_MPS_HIGH_WATERMARK_RATIO=0.0 to disable upper limit for memory allocations (may cause system failure)."

Devise strategies to manage the image inputs and global memory limits, in order to prevent overflows. That may mean serializing tasks somehow or identifying other optimizations in model implementation, especially on Apple Silicon.

This is a substantial set of changes. Let me work through them systematically.Now let me look at the full template:Now I have a full picture. This is a large set of changes. Let me rebuild all the modified files. The changes span every file in the project. I'll create them one at a time.This is a massive rewrite. Let me build all the files systematically. I'll create each one with all the new features integrated.Now the heavily revised OCR engine with memory management, persistent model, custom tags, and metadata inference:Now the Flask server — heavily revised to support persistent model, metadata editing, custom tags, and the new workflow:Now the big one — the completely rewritten HTML template with the new workflow (upload → process → confirm metadata → save), custom tags UI, language autodetect, and model loading status:Now generate the app icon and fix the setup script (renamed to TEI Pipeline, icon included, fix the bouncing/cancel-sign issues):

[Interrupted due to context limit]

> I can see from the transcript that I was already in the middle of implementing these changes. Let me check what's been completed and what still needs to be done.
>
> Most files are created. I still need the setup_app.py (renamed to TEI Pipeline, with icon embedding and the fixes for the cancel-sign/bouncing issues) and a README. Let me verify the existing files are complete, then create the missing ones.
>
> Verify all Python files parse
>
> Test core modules
>
> Good. Now create the setup_app.py with all fixes:
>
> 1. **Cancel sign** — macOS shows this when the executable isn't properly signed or when `LSUIElement` confuses the Finder. The fix: ensure the app has a proper `CFBundleIconFile` entry and use `LSApplicationCategoryType`. Also, the cancel sign often appears because the shell script launcher isn't recognized as a valid executable by Gatekeeper — we need to sign it with ad-hoc codesigning.
> 2. **Long bouncing** — The app bounces in the Dock until it registers with the window server. A shell script never does this. The fix: use `LSBackgroundOnly` or better, use AppleScript/`open -a` to immediately signal the Dock, and add `LSUIElement` handling. The real solution: tell the Dock we're ready by using a tiny AppleScript wrapper that opens the browser immediately while the server starts in background.
> 3. **Icon** — embed the .icns file (built from the iconset) and reference it in Info.plist.
>
> Complete setup_app.py: renamed to TEI Pipeline, embeds icon as .icns, fixes cancel-sign (ad-hoc codesign) and bouncing (AppleScript LSUIElement trick), all memory management env vars

Complete setup_app.py: renamed to TEI Pipeline, embeds icon as .icns, fixes cancel-sign (ad-hoc codesign) and bouncing (AppleScript LSUIElement trick), all memory management env vars

Complete setup_app.py: renamed to TEI Pipeline, embeds icon as .icns, fixes cancel-sign (ad-hoc codesign) and bouncing (AppleScript LSUIElement trick), all memory management env vars

Now the README:

Now verify everything together:

Copy everything to outputs

You'll also need the `icon.iconset/` folder (9 PNG files at various sizes). Here they are:

Here's a summary of everything that changed:

Renamed to "TEI Pipeline" throughout (app name, bundle ID, UI title, all references).

New features:

- **Metadata inference** — after OCR, the VLM is asked to extract title/author/date/publisher/place/language from the first page. These populate a new Step 3 ("Confirm Metadata") form. The inferred values go into the initial TEI output. Editing them and clicking "Apply & Regenerate XML" calls `/api/update-metadata`, which regenerates the XML from the stored OCR results without re-running inference.
- **Language auto-detect** — the dropdown defaults to "Auto-detect"; the VLM infers the ISO 639 code during metadata inference. The inferred code appears in the metadata confirmation form.
- **Custom tags** — an expandable "Custom Tags (advanced)" section lets users add tag name + description pairs before processing. These get appended to the VLM prompt as `[TAG_NAME]...[/TAG_NAME]` markers. They'll appear in TEI output as `<seg type="tagname">` elements.
- **Persistent model** — the model loads in a background thread at server startup and stays in memory across document uploads. The UI shows a status badge ("Model loading..." → "Model ready").
- **App icon** — a blue document-with-angle-brackets icon, built as an iconset and converted to .icns during setup.

Bug fixes:

- **Cancel sign on icon** — the setup script now runs `xattr -cr` (removes quarantine) and `codesign --force --deep --sign -` (ad-hoc codesigning) on the .app bundle.
- **Long dock bouncing** — the launcher now waits up to 45s for the server and opens the browser as soon as it's ready, rather than a fixed sleep. `LSUIElement` is set to `False` so macOS treats it as a normal app.

MPS memory overflow fixes (the big one):

- `PYTORCH_MPS_HIGH_WATERMARK_RATIO=0.0` set in both the launcher and in `ocr_engine.py` — removes the artificial 30 GB ceiling
- Images capped at **1280px** per side on MPS (down from 2048) — this dramatically reduces vision token count
- `torch.mps.empty_cache()` + `torch.mps.synchronize()` + `gc.collect()` between every page
- Input/output tensors explicitly `del`eted after each inference call
- 3B model option available as a fallback for tight memory situations

The program is working and generates text. There are a few more issues to resolve. First I will present the small issues to fix. Produce code at an intermediate step to incorporate the changes, but do not yet present it in the chat. Then I will present the large issues, regarding program performance, afterward

App Icon: The cancel symbol still appears over icon when installed. But it is removed when Finder restarts. Figure out whether there is a way to safely import the app to the Applications folder without the cancel symbol. Also, generate a new icon for the program with letters "TEI" and a simplified pipe like ones in Super Mario Bros. Include just a single PNG file in the program folder.

Exit Program: Closing the program window appears to quit the program. However the Python process continues running afterward. End the Python process on quitting too.

TEI Schema: Try to streamline the <teiHeader> information. For example, if the TEI guidelines permit you to omit the <fileDesc> section, then simply put all bibliographic metadata into the <sourceDesc> section. It is more valuable to know information about the source, than it is about the file itself.

Let me work through each of these issues. First, let me review the current tei_schema.py to understand the header structure, and check the TEI guidelines.Now let me check the current setup_app.py launcher for the exit issue, and look at the TEI P5 guidelines regarding `<fileDesc>` being required:Now let me also review the server to understand shutdown behavior:Now I have a clear picture of all three issues. Let me research the TEI P5 requirements for `<fileDesc>`:

The TEI P5 guidelines state that `<fileDesc>` is **required** within `<teiHeader>` — it's the only mandatory child. And within `<fileDesc>`, `<titleStmt>`, `<publicationStmt>`, and `<sourceDesc>` are all required. So I can't eliminate `<fileDesc>`, but I can make it minimal (just the required stubs) and put the real bibliographic detail in `<sourceDesc>` using a proper `<bibl>` element.

Now let me generate the new icon and make all changes:Good. Now let me implement all three fixes. I'll modify the files in place:

1. Icon / cancel symbol fix — The cancel symbol appears because macOS quarantines apps created by non-notarized processes. `codesign` alone isn't enough; we also need to remove the `com.apple.quarantine` xattr *after* everything is written, and the icon needs to be `.icns` not just `.png` (macOS requires `.icns` for proper icon display without the cancel badge). The setup script should also try to use `iconutil` to build `.icns` from the iconset, and only include icon.png in the source directory (the iconset gets built by setup_app.py itself).

2. Exit/shutdown — Add a `/api/shutdown` endpoint and a trap in the launcher script so killing the .app kills the Python process. Also add a browser `beforeunload` listener that calls shutdown.

3. TEI Schema — Keep `<fileDesc>` (required by TEI P5), but make `<titleStmt>` and `<publicationStmt>` minimal stubs. Move all real bibliographic info into `<sourceDesc>` using `<bibl>`.Now add the shutdown endpoint to server.py and the browser-side exit handling:Now add the browser-side shutdown trigger to the HTML. When the user closes the tab/window, fire a beacon to `/api/shutdown`:Now update setup_app.py: fix the icon approach (generate iconset from icon.png at install time, try iconutil, use xattr -d specifically for the quarantine attribute), and add a trap to the launcher so killing the .app kills the server:Wait — I have a problem with f-string escaping in that file. The shell script inside the f-string uses `{{` for literal braces, but I also have Python f-strings with `{{` that shouldn't be doubled. Let me check:Wait, let me look more carefully — the print statements in main() use `{{APP_NAME}}` and `{{SCRIPT_DIR}}` which would render as literal `{APP_NAME}` in output since they're inside an f-string. But those aren't in f-strings — they're regular strings. Let me check:This is the problem. Those `{{APP_NAME}}` etc. should be `{APP_NAME}` in regular f-strings. The `{{}}` are only needed inside the shell script string (which is itself inside an f-string). But outside the shell script, in `main()`, these are standalone f-strings and the `{{}}` will print literal braces. The shell script part on line 160 (`cleanup() {{`) is correct. But lines 194+ are wrong.

> The root cause: the entire file isn't inside a single f-string — each `print(f"...")` is its own f-string. So `{{SCRIPT_DIR}}` would print `{SCRIPT_DIR}` literally. I need to fix this:The header is now streamlined: minimal `<titleStmt>` and `<publicationStmt>` stubs, all real metadata in `<sourceDesc><bibl>`, and language in `<profileDesc>`.
>
> Let me also clean up the stale icon files that are no longer needed (we just ship icon.png):Final verification of everything:All three fixes verified. Holding off on presenting files as you requested — ready for the larger performance issues.

The OCR'ed text is accurate, however the markup tags are not always correctly applied and there are many artifacts left by the process of generating text and parsing it.

I have uploaded with this message an XML file as it was "generated" by the current program and a version of it that was "cleaned" by hand. (Also uploaded is the original PDF.) The generated file contains many artifacts of the OCR to TEI process such as the word "plaintext" and some of the vision model's own tags (like "[/PARAGRAPH]"). These artifacts indicate a problem with the way generated texted is parsed afterward. Use the cleaned file to guide the revision process.

Note that in a generated file for a second PDF, the a good deal of the vision model's raw text preceded the parsed text. In a generated file for a third PDF, every line began with "[" and ended with "]". This artifact suggests a problem with the way the prompt is written. (Perhaps it contains too many brackets, priming the model to generate them.)

There are three possible causes of errors:

(1) The way the vision model is implemented or optimized. For example, resizing images may produce challenges for the vision models.

(2) The way the prompt is written for the vision model. For example, the instructions may be too complex or it may be useful to instruct the model to use the TEI tags directly.

(3) The way the generated text is parsed afterward. For example, the regex matching may need to be more robust or it may be useful to instruct a small language model to parse the generated text (but do not forget the memory limits on the program).

Identify likely causes of the markup artifacts and the most effective solutions that will resolve them. It may not be clear which solution will be most effective. Consider addressing each cause of error in different ways, and propose the solutions that are most likely to resolve the errors. Report your assessment: which artifacts are addressed by each solution, how likely are they to be resolved, and any tradeoffs with model performance or efficiency.

> Let me carefully examine the generated vs. cleaned XML files to identify the specific artifacts, then trace them back through the pipeline.Now let me also look at the OCR engine prompt and the tei_generator parser to trace how each artifact arises:Now let me

look at the OCR prompt more carefully, and also look at the project knowledge files for reference:Now I have a thorough picture. Let me catalog every artifact I see in the generated file and trace each one to its cause.

Here is my systematic assessment:

Artifact Inventory

Comparing the generated XML to the cleaned XML and your descriptions of other PDFs, I identify these distinct artifact categories:

A. `` ```plaintext `` / ``` `` wrappers (lines 34, 44-45, 51, 59)

The VLM is wrapping its output in markdown code fences. This is a **prompt issue** (Cause 2): Qwen2.5-VL, trained heavily on code/markdown tasks, defaults to wrapping structured output in code blocks. The current prompt says nothing to prevent this.

B. Residual `[/PARAGRAPH]` tags in output (lines 38, 40, 42-43)

The parser searches for `[PARAGRAPH]` as a *standalone* split marker but doesn't strip the **closing** tag `[/PARAGRAPH]`. The VLM sometimes generates `[PARAGRAPH]...[/PARAGRAPH]` as a pair (treating it like the other tags), but the regex only looks for the opening marker. This is a **parser issue** (Cause 3).

C. `[PAGE_NUMBER] 195 [PAGE_NUMBER]` at line 83

This is a malformed tag — the VLM used `[PAGE_NUMBER]` twice instead of `[PAGE_NUMBER]...[/PAGE_NUMBER]`. The parser regex expects `\[PAGE_NUMBER\](.*?)\[/PAGE_NUMBER\]` and doesn't match it, so it passes through raw. **Cause 2** (prompt ambiguity about which tags are paired) and **Cause 3** (parser not robust to malformed tags).

D. Running header/page number duplicated in body text (lines 52-53, 63, 75)

The VLM outputs `The Heresy of Paraphrase 193` or `194 THE WELL WROUGHT URN` as plain text at the start of pages, in addition to sometimes also marking them with `[RUNNING_HEADER]` or `[PAGE_NUMBER]` tags. The parser extracts the tagged versions but the untagged duplicate remains as body text. **Cause 2** (prompt doesn't tell the model to tag all headers/page numbers and not repeat them as text) and **Cause 3** (no post-processing to detect/remove these duplicates).

E. First page content dumped into `<front>` (lines 37-45)

The `_has_front_matter` heuristic incorrectly classified the entire first page as front matter, including three full paragraphs. The cleaned version correctly shows only the chapter heading in `<front>`, with the body text in `<body>`. **Cause 3**: the heuristic checks if the first page "is short and has a heading" — but this page has a heading *and* substantial body text, so it fails.

F. Lines beginning with `[` and ending with `]` (reported for third PDF)

This strongly suggests the model is being **primed** by the bracket-heavy prompt to generate bracket-delimited output for everything. When the prompt contains 20+ lines of `[TAG]...[/TAG]` examples, the model sometimes generalizes and wraps every line in brackets. **Cause 2**: the prompt format itself creates this tendency.

G. Raw VLM text preceding parsed text (reported for second PDF)

This sounds like the model generated preamble/explanation before the transcription. The parser then failed to separate it. **Cause 2** (prompt doesn't explicitly say "do not output anything other than the transcription") and **Cause 3** (no stripping of preamble).

Proposed Solutions

Solution 1: Rewrite the prompt to use simpler, less bracket-heavy markers

Addresses: A, C, D, F, G

The current prompt uses 20+ lines of `[TAG]...[/TAG]` notation. This creates two problems: it's too complex for reliable generation, and the bracket-heavy format primes the model to generate brackets everywhere.

Change: Use XML-like tags directly (e.g., `<head>`, `<p>`, `<note>`) or simpler delimiters (e.g., `---HEADING---`). Better yet, use a two-phase approach: first ask for plain transcription, then ask for structural annotation. But the single-pass approach is better for memory.

My recommendation: Switch to a much shorter prompt that asks the model to output in a lightweight markup that's less ambiguous. Use `#` for headings, blank lines for paragraph breaks, and only a handful of explicit tags for things that can't be inferred structurally (italic, footnote, page number). This reduces the chance of hallucinated tags and bracket priming.

Also add explicit instructions: "Do NOT wrap output in code fences. Do NOT add explanations or preamble. Start with the transcribed text immediately."

Likelihood of resolving artifacts: High for A, F, G. Medium for C, D.

Tradeoffs: Simpler prompts may miss some structural features (like catchwords or column breaks). But these features were already unreliably detected.

Solution 2: Robust post-processing / cleanup layer in the parser

Addresses: A, B, C, D, G

Add a `_clean_raw_output()` method that runs before any structural parsing:

1. Strip markdown code fences (`` ```plaintext ``, `` ``` ``)
2. Strip `[/PARAGRAPH]` closing tags (convert `[PARAGRAPH]...[/PARAGRAPH]` to `[PARAGRAPH]`)
3. Fix malformed tags: `[TAG] text [TAG]` → `[TAG] text [/TAG]`
4. Strip any text before the first structural marker (preamble removal)
5. Detect and remove duplicate running headers/page numbers that appear as untagged plain text at the start of pages (heuristic: if the first line of a page matches a `[RUNNING_HEADER]` value or looks like `Title 123`, strip it)

Likelihood: Very high for A, B. High for C, G. Medium for D (heuristic-dependent).

Tradeoffs: Minimal. This is pure defensive cleanup and can't make things worse.

Solution 3: Fix the front-matter heuristic

Addresses: E

The current heuristic is too aggressive — it calls the entire first page "front matter" based on the presence of a heading and page length. The fix: only classify as front matter if the page has a heading but *very little* body text (e.g., `[PARAGRAPH]` count ≤ 1 and total text after heading is < 200 chars). A title page typically has a title, maybe an author, and little else.

Likelihood: High. This is a straightforward logic fix.

Tradeoffs: None. More conservative heuristic means fewer false positives.

Solution 4: Don't downscale images for MPS

Addresses: potentially some OCR quality issues

You mentioned this as cause (1). The current code caps images at 1280px on MPS. If the original PDF is rendered at 2048px and then downscaled, some fine detail (small print, footnotes, tight kerning) could be lost. However, looking at the generated output, the **text transcription is accurate** — the artifacts are all structural markup problems, not OCR accuracy problems. So image scaling is probably not the cause here.

Recommendation: Keep the 1280px cap for memory safety but expose it as a user-configurable option. This is low priority relative to the prompt and parser fixes.

Likelihood of fixing the observed artifacts: Low (the text itself is correct).

Priority Ranking

Priority	Solution	Artifacts Fixed	Confidence	Effort
1	**Solution 2**: Robust cleanup layer	A, B, C, D, G	Very high	Low
2	**Solution 1**: Simplified prompt	A, C, D, F, G	High	Medium
3	**Solution 3**: Fix front-matter heuristic	E	High	Low
4	Solution 4: Image scaling option	(quality edge cases)	Low	Low

Solutions 1 and 2 together address nearly all the observed artifacts. Solution 2 is defensive (catches whatever the model produces) while Solution 1 is preventive (reduces the likelihood of bad output in the first place). They're complementary and should both be implemented.

Shall I proceed with implementing these changes?

Before implementing, one small question: is the <sourceDesc> tag required in TEI?

Yes. In TEI P5, `<sourceDesc>` is a required child of `<fileDesc>`, which is itself the only required child of `<teiHeader>`. The three mandatory children of `<fileDesc>` are `<titleStmt>`, `<publicationStmt>`, and `<sourceDesc>`. You cannot omit any of them and produce valid TEI.

That said, `<sourceDesc>` can be minimal — a single `<p>` with a text description satisfies the requirement. So the current approach of putting a `<bibl>` with the real metadata inside `<sourceDesc>` is the right place for it. The schema we have is already about as lean as TEI P5 allows.

Implement Solution 1 (Simplified prompt), Solution 2 (Robust cleanup layer), and Solution 3 (Fix front-matter heuristic). Be sure that Solution 2 is coordinated with any changes to the expected outputs from Solution 1. Present the code for the new program.

Let me start by reviewing all the files I need to modify, then implement all three solutions in a coordinated way.Now I have the full picture. Let me implement all three solutions. The key design decisions:

Solution 1 (Prompt): Use a much simpler markup system. Instead of 20+ bracket tags, use a small set of natural-language-style markers that are less likely to prime the model. Use `## HEADING ##` style for headings, `---` for page elements, and keep paragraph breaks as blank lines. Explicitly forbid code fences and preamble.

Solution 2 (Cleanup): Add a `clean_raw_output()` function that runs on every page's text before the parser sees it. This strips code fences, normalizes/fixes malformed tags, removes preamble, and strips duplicate headers.

Solution 3 (Front-matter): Rewrite the heuristic to only classify as front matter if the page has a heading but almost no body text (paragraph count ≤ 1 AND body text after heading < 300 chars). Also, when a page *is* front matter, split the heading portion from the body portion rather than dumping the whole page into `<front>`.Now the big one — the completely rewritten TEI generator with Solutions 2 and 3:Now let me run thorough tests, including testing the cleanup layer against the actual generated output from the Brooks PDF:The bracket-stripping regex only catches `[/UPPERCASE]` and `[UPPERCASE]` patterns. The artifact `[This is some text]` has lowercase and spaces inside — that's a different pattern. Let me fix the orphan bracket cleanup to also handle these:The prose pipeline test failed. Let me debug:I see — the first page's body text went into front-matter's modified page, but there's no `<pb>` for it in the body, and the body starts at page 2. The issue is that when the front-matter extractor splits the heading from the body text, the body text of page 1 is correctly included in `body_pages`, but the `_build_prose_body` processes it with a page break for label "1". Let me check the actual assertion failure:

The problem is that `<fw type="header">` and `<fw type="pageNum">` are only appearing for page 2. Page 1's RH/PN were already consumed by `clean_raw_output` but then the front-matter extraction stripped them because the body_text is text after headings. Let me trace more carefully:Good — `clean_raw_output` preserves RH/PN lines. Now the front-matter extractor removes headings and RH/PN to measure body text length. The body_text is "The ten poems..." which is > 200 chars, so it returns the headings in front and creates a modified first page with the body text. But the modified first page has the body_text WITHOUT the RH/PN lines (they were stripped during measurement). That's the bug — the modified first page should retain the full text minus just the headings.I see — page 1's fw is missing because when the front-matter extractor creates the modified first page, the `_build_prose_body` adds a `<pb n="1">` followed by parsing. But `_parse_prose_page`calls `_extract_forme_work` which should find the RH/PN lines. Let me check what the modified first page text looks like:Good — the remaining text has RH/PN lines. Now let me check if `_parse_prose_page` handles them. The issue is that the `_extract_forme_work` method's duplicate-header detection might be eating the first line after RH/PN:The fw elements ARE being created from page 1's body text. The problem is that the test assertion `'fw' in xml.split('<body>')[1]` is looking in the body section, and the fw elements ARE there. Let me re-run the failing test more carefully:The body text "The ten poems" is missing entirely! And PN 192 is not in the body. Let me see the full XML:The modified first page text is not appearing in the body at all! The front-matter extraction returns the modified page, but `_build_prose_body` doesn't process it. Let me trace: the `generate_tei` method gets `body_pages` from `_extract_front_matter`, and passes them to `_build_prose_body`. But I see only page 2 in the output. The issue must be that `body_pages` only contains page 2. Let me check:Found it. `body_pages` only has page 2. The modified first page was NOT included. Let me look at the code:

```
remaining_text = re.sub(r'##\s*.*?\s*##', '', first_text).strip()

body_only = ... # measure

if len(body_only) < 200:

    return headings, pages[1:]  # ← this path is taken
```

The body_only length is "The ten poems that have been discussed.\n\nOn reflection, however." which is only ~77 chars! That's < 200, so it's classified as a "true title page" and pages[1:] is returned. The 200-char threshold is too high for this case. Let me reconsider — the issue is that my test text is artificially short. With real text it would be much longer. But the heuristic needs to work for both short and long tests.

The real fix: the threshold should be about the number of paragraph blocks, not character count. A title page has 0-1 paragraphs. A chapter opening has 2+.All tests pass. Now let me copy everything to outputs and present:Here's a summary of what changed in the three key files:

`ocr_engine.py` — Solution 1 (Simplified Prompt):

- Replaced the 20+ `[TAG]...[/TAG]` markers with a much lighter format: `## heading ##` for headings, blank lines for paragraphs, `RH:` / `PN:` for running headers/page numbers, `_italic_` / `**bold**` / `FN{...}` / `SC{...}` for inline elements
- Added explicit anti-code-fence and anti-preamble instructions at the end of every prompt
- The prompt is about half the length, with far fewer bracket characters, which should eliminate the bracket-priming artifact

`tei_generator.py` — Solution 2 (Cleanup Layer) + Solution 3 (Front-Matter Fix):

- New `clean_raw_output()` function runs on every page before parsing. It strips code fences, preamble text, normalizes all legacy `[TAG]...[/TAG]` markers into the new format, fixes malformed tags (like `[PAGE_NUMBER] 195 [PAGE_NUMBER]`), strips orphaned bracket tags, and removes bracket-wrapped lines
- The cleanup layer is **backward compatible**: if the model still outputs old-style bracket tags (which it might on occasion), they get normalized to the new format before the parser sees them
- New `_extract_front_matter()` uses paragraph count (≤1 block AND <300 chars) instead of the old "short page + has heading" heuristic. When a page has a heading AND substantial body text, only the headings go to `<front>` — the body text (including its RH/PN lines) is returned as a modified first body page
- All parsers (prose, poetry, drama, manuscript) rewritten to work with the new marker format
- New `_append_text()` helper eliminates the repeated text-insertion boilerplate

The test suite covers all artifact categories from the analysis (A through G) plus the Brooks PDF simulation, all genre types, and custom tags.

The Brooks file is much better. I have uploaded the output from the new version of the program. The remaining issues deal with the running header: The chapter title in the running header is wrapped in <p> tags, rather than <fw> or similar. Also, the header with title and page number seems to be missing from the third image <pb n="3">. It is possible that the parser removed it or that it was not generated at all by the vision model. Also, the chapter title "## Chapter Eleven THE HERESY OF PARAPHRASE" is wrapped in <p> even though it is clearly preceded by special markup tags.

The title page heuristic needs to be revised. It now interprets an actual title page as poetry. Be careful not to go too far in the other direction. Poetry still needs to be recognized as such. Image of title page and XML output attached to this message as well.

Let me examine the uploaded files carefully to diagnose each issue.

Good diagnostic data. Let me trace each issue to its cause.

Issue 1: `## Chapter Eleven THE HERESY OF PARAPHRASE` wrapped in `<p>` — The VLM output the heading with `##`markers, but the `##` was not on its own line — it was embedded in a paragraph block. The `_parse_prose_page` method splits by `(## ... ##)` regex, but if the `##` text doesn't have blank lines before/after it, it gets merged into a paragraph. The regex split `re.split(r'(##\s*.*?\s*##)', text)` should work, but the VLM likely put it on a line without surrounding blank lines. Also the `_extract_forme_work` duplicate-header detector may not be stripping it.

Issue 2: Running header "The Heresy of Paraphrase" as `<p>` on pages 2 and 4 — The VLM output the running header as a plain line without `RH:` markup. The parser has a duplicate-header heuristic but it only catches lines that are short, contain a number, AND have >30% uppercase. "The Heresy of Paraphrase" has no number and isn't mostly uppercase, so it passes through as a `<p>`.

Issue 3: Page 3 missing header entirely — Either the VLM didn't generate it, or the duplicate-header heuristic stripped it (page 3 has "194 THE WELL WROUGHT URN" which IS short + has number + is uppercase — exactly the pattern it strips).

Issue 4: Title page detected as poetry — The genre detection counts short lines. A title page with centered text produces many short lines, triggering the poetry heuristic. Need to exempt single-page documents or the first page from the poetry line-count check.

Let me fix all four issues:

Now I understand all the issues clearly. Let me implement the fixes:

Fix 1 (heading `##` not parsed): The `##` regex uses `\s*` which doesn't match newlines by default. If the VLM outputs `## Chapter Eleven\nTHE HERESY OF PARAPHRASE ##` with a newline inside, the regex `##\s*.*?\s*##` won't match because `.` doesn't match `\n` by default. Need `re.DOTALL` flag in the split.

Fix 2 (running header as `<p>`): The duplicate-header heuristic is too narrow. Need to also detect the first line of a page if it's short and matches a known title pattern (no number required). Better approach: detect the first line as a running header if it's short (< 80 chars), is on its own line (followed by blank line), and doesn't look like body text.

Fix 3 (page 3 missing header): The heuristic catches "194 THE WELL WROUGHT URN" and strips it — but then doesn't create an `<fw>` element for it. Need to create the `<fw>` when stripping.

Fix 4 (title page as poetry): The poetry heuristic fires on title pages because they have many short centered lines. Fix: only count lines from pages that have >1 page (skip single-page documents from the poetry heuristic), and also check that the page doesn't look like a title page (has `##` headings or metadata markers).

Now fix the heading regex to handle multi-line headings (Fix 1):

Now fix the genre detection (Fix 4) — don't count title pages toward poetry score:

Also fix the heading split in poetry parser to handle multiline `##`:

Now let me also fix the front-matter detection for title pages. The title page detection currently requires `##` headings, but the Pound/Cathay page has no `##` markers — the VLM just outputs the centered text as short lines. The title page needs to be detected even without `##` markers:

Now run the tests:

Test all fixes

The title page heuristic (Path B) is too aggressive — "The Heresy of Paraphrase\n\nto being exhaustive of periods." is only 2 lines, both short, and total < 500 chars, so it triggers the title page detection. The body text "to being exhaustive of periods." starts lowercase, which should be a signal that it's NOT a title page. Let me tighten the heuristic:

Tighten Path B title page detection: require most lines to start with uppercase, and require at least 3 lines

Let me check where the poetry test is failing and fix it:

The poetry page has only 2 blocks and len < 1000, so it's being skipped as a "title page" by the poetry detection. The filter `len(blocks) <= 2 and len(text) < 1000` is too aggressive — it correctly excludes title pages but also excludes short poems. I need a better way to distinguish title pages from poetry pages. A title page is characterized by very short lines with lots of whitespace between blocks. Poetry has regular-length verse lines in consistent stanzas.

Fix poetry detection: better title page vs poetry discrimination — title pages have very short avg line length and few lines per block, while poetry has longer lines and more lines per block

All passing. Now let me also test the Brooks page 4 case — the running header "The Heresy of Paraphrase 195" where the em-dash continuation starts with `—` which is a special check in my code:

The header is correctly extracted as `<fw>` with the page number split out, and the em-dash continuation is preserved as body text. Now let me copy to outputs:

Here's what changed relative to the previous version, addressing each reported issue:

Issue 1 — `## Chapter Eleven THE HERESY OF PARAPHRASE` in `<p>`: The heading split regex now uses `re.DOTALL` so `## ... ##`markers that span newlines are correctly matched. Multi-line headings are normalized to single-line text in the `<head>`element.

Issue 2 — Running header "The Heresy of Paraphrase" as `<p>`: Rewrote `_extract_forme_work` with a smarter untagged-header detector. It now examines the first 1-2 lines of each page and classifies them as `<fw>` if they're short, standalone (followed by blank line), and title-case/uppercase. It also splits combined lines like "194 THE WELL WROUGHT URN" into separate `<fw type="pageNum">` and `<fw type="header">` elements. Lines starting lowercase or with em-dashes are correctly left as body text.

Issue 3 — Page 3 missing header: The old heuristic stripped "194 THE WELL WROUGHT URN" but didn't create `<fw>`elements. Now it always creates `<fw>` elements when extracting untagged headers.

Issue 4 — Title page misdetected as poetry: Rewrote the genre detection to distinguish title pages from poetry by checking average line length and lines-per-block. Title pages have very short lines (avg < 35 chars) and few lines per block (< 4), while poetry has longer verse lines and 4+ lines per stanza. Also added a separate Path B in front-matter detection that catches title pages even without `##` markers, by looking for pages that are all short uppercase lines with no prose paragraphs.