ChatGPT

Offline LaTeX PDF Conversion Pipeline (Fedora Linux)

Overview: This report outlines a fully local pipeline to convert PDFs (including scanned or complex scholarly PDFs) into high-fidelity LaTeX. The solution uses **100% open-source tools** and runs entirely offline on Fedora Linux, leveraging the GPU (RTX 5090, 32 GiB VRAM) for acceleration. We break down the pipeline into stages—document layout analysis, content extraction (text, equations, tables, figures), LaTeX assembly, and validation—highlighting tools like *Docling*, *Nougat*, and *pix2tex* and how they integrate. Installation instructions, performance optimizations for the given hardware, debugging tips, and optional enhancements (like local RAG search and equation OCR fallback) are provided.

Pipeline Components and Tools

- **Document Layout & Parsing – *Docling*:** IBM's Docling library parses PDFs (and other formats) into a structured representation. It performs advanced layout analysis (detecting headings, paragraphs, tables, formulas, images, etc.) and supports OCR for scanned documents [1]. Docling can run fully offline (no cloud) and is optimized for local execution [2]. We use it to segment the PDF into logical blocks (text blocks, equations, figure regions, etc.) before extraction.

- **Vision OCR for Text & Math – *Nougat*:** Meta's Nougat is a visual Transformer model that performs OCR on academic PDFs, outputting marked-up text with LaTeX for math and tables [3]. It converts each page image to *Mathpix Markdown* (a Markdown variant preserving LaTeX). Nougat excels at reading scientific text, equations, and even tables from page images entirely on GPU, making it suitable for both scanned and digital PDFs. We will use Nougat to extract text and math content with high fidelity, taking advantage of the RTX 5090 for fast inference.

- **Equation OCR – *pix2tex*:** For isolated equation images or as a fallback, *pix2tex* (LaTeX-OCR) can convert an image of a math formula directly into LaTeX code [4] [5]. This tool uses a ViT-based model to accurately recognize mathematical notation. In our pipeline, pix2tex serves as a specialized *math OCR agent* [6], ensuring that even tricky displayed equations are captured as editable LaTeX (not just as images).

- **Additional Utilities:** We also leverage standard tools for completeness:

- *Tesseract OCR* (with Python binding `tesserocr` or via Docling) for text OCR as needed (especially if not using Nougat for all text).
- *PDF rendering libraries* like PyMuPDF (MuPDF) to extract images (figures or formula crops) and coordinate information from PDFs.
- *LaTeX environment* (TeX Live, `latexmk`) to compile and verify the generated `.tex` files.
- *Table extraction models:* (optional) Docling's layout parser can identify table structure; if needed, a model like Microsoft TableTransformer could be used for complex tables, though Docling's output or Nougat's Markdown tables are often sufficient.

- *Vector DB & LLM (optional):* For Retrieval-Augmented Generation (RAG) enhancements, we could integrate a local vector database (e.g., FAISS or Chroma) of previous LaTeX documents and a local LLM to help correct or fill gaps in OCR output. This is an advanced add-on discussed later.

Installation and Setup (Fedora)

Before implementing the pipeline, prepare the system with all required dependencies:

1. System Packages: Install system libraries for OCR and LaTeX compilation. This includes Tesseract (for OCR engine and language data) and TeX Live (for compiling and verifying output). On Fedora, use `dnf` to install these:

```
sudo dnf install -y tesseract tesseract-devel leptonica-devel tesseract-
langpack-eng \
                    tesseract-osd poppler-cpp texlive-scheme-full latexmk
# Set Tesseract data path (if not in default location):
export TESSDATA_PREFIX="/usr/share/tesseract/tessdata/"
echo "TESSDATA_PREFIX=$TESSDATA_PREFIX"
```

The above ensures Tesseract OCR is available (with English language data) [7] and a full TeX Live distribution (including `latexmk`) is installed for LaTeX processing. We also include Poppler (`poppler-cpp`) which some PDF tools use for rendering (Docling/PyMuPDF).

2. NVIDIA Drivers and CUDA: Make sure NVIDIA drivers and CUDA are installed to utilize the RTX 5090. On Fedora, this typically means installing NVIDIA's proprietary driver (if not already) and CUDA toolkit libraries. (This step may vary, but ensure that `nvidia-smi` can see the GPU and that CUDA is accessible for frameworks like PyTorch.)

3. Python Environment: Set up a Python 3.10+ environment (virtualenv or Conda) for the pipeline tools. Install PyTorch with CUDA support, Docling, Nougat, and pix2tex:

```
# Activate virtual environment first (if using venv or conda)
pip install torch torchvision torchaudio --extra-index-url https://
download.pytorch.org/whl/cu118

# Install Docling with extras for VLM and Tesseract OCR support
pip install "docling[vlm,tesserocr]"

# Install Nougat OCR (Meta's PDF parser)
pip install nougat-ocr

# Install pix2tex (LaTeX-OCR for equations)
pip install pix2tex

# (Optional) Other OCR engines if desired:
```

```
pip install easyocr rapidocr onnxruntime  # if using EasyOCR or RapidOCR as
alternatives
```

The above installs: - **PyTorch** (with CUDA 11.8 in this example – adjust version as appropriate for RTX 5090 and latest PyTorch). This is a prerequisite for both Nougat and pix2tex. - **Docling** with `vlm` and `tesserocr` extras: the `vlm` extra pulls in visual layout model dependencies (like Transformer-based layout analyzers), and `tesserocr` provides a direct Tesseract binding [8] [9]. Ensure that the system Tesseract is installed as done in step 1, so Docling can link to it. - **Nougat** (`nougat-ocr` package): the pipeline will download the Nougat model checkpoint on first use (choose the appropriate model size as needed). - **pix2tex**: installs the LaTeX-OCR model; the first run will auto-download its pretrained weights.

4. Verify Installations: After installation, verify each tool:

```
# Check Docling version and basic usage
docling --version
docling --help  # should display CLI options

# Check Nougat CLI is available
nougat --help

# Check pix2tex
pix2tex --help  # (this opens the CLI usage for pix2tex)
```

Also, test that `latexmk` is found (e.g., `latexmk -v`) and that compiling a simple LaTeX file works, to ensure TeX Live is functional.

Step 1: Document Layout Analysis and Segmentation

The first stage uses **Docling** (with a visual layout model) to analyze the PDF's structure **without relying on external APIs or embedded text**. This gives us a *master plan* of the document: all text blocks, headings, images, tables, and formulas with their bounding boxes and reading order [10]. By performing a multimodal layout analysis up front, we can treat each region appropriately before OCR or LaTeX generation.

Using Docling: We can invoke Docling via its Python API or CLI. The Python approach offers more control:

```
from docling.document_converter import DocumentConverter
from docling.datamodel.pipeline_options import PipelineOptions,
TesseractOcrOptions

# Configure Docling to do layout analysis (and OCR if needed)
opts = PipelineOptions()
opts.do_ocr = True               # enable OCR for scanned PDFs
opts.ocr_options = TesseractOcrOptions()  # use Tesseract via tesserocr for OCR
opts.do_layout_analysis = True   # ensure layout model is used (should be true
```

```
by default)
# Optionally use VLM (visual language model) for layout for better accuracy:
opts.pipeline = "vlm"
opts.vlm_model = "granite_docling"  # using IBM Granite model on HF for
layout+OCR

converter = DocumentConverter(pipeline_options=opts)
doc = converter.convert("input.pdf").document  # Process the PDF into a
DoclingDocument
```

In this configuration, Docling will: - Use a **visual layout model** (like *GraniteDocling*) to segment each page into regions (headings, paragraphs, tables, formulas, images, etc.). - Perform OCR on each region if needed. For a **born-digital PDF**, Docling will by default attempt to extract the text without OCR and only fall back to OCR for regions it can't get text for (like scanned images). For **scanned PDFs**, `do_ocr=True` ensures text is extracted via Tesseract in each region. (You could also substitute `EasyOcrOptions` or `RapidOcrOptions` to use those engines [9].)

Layout Output: Docling's result (`doc`) contains a structured representation of the document. Each page is composed of a list of blocks with attributes (text content if available, bounding box coordinates, type of block, etc.). We can export this to JSON or Markdown:

```
# Export to a JSON with all layout and content (for inspection or debugging)
layout_json = doc.export_json()
# Or export to Markdown (which will include text content and some markup for
tables/images)
md_text = doc.export_to_markdown()
```

At this stage, we have *segmented the PDF*: - **Text blocks** (paragraphs, section headers, etc.) with their text (for digital PDFs) or empty content (for scanned, until OCR is applied). - **Image regions** (figures, diagrams). - **Table regions** (with structure detected; Docling can often decode tables into CSV or Markdown table format). - **Formula regions** (likely identified as figures or formula blocks; with `do_formula_enrichment=True` Docling tries to preserve formula LaTeX if the PDF natively contains TeX, but typically in PDFs formulas are not stored as TeX text, so an OCR step is needed) [11].

Why this step matters: By identifying content type upfront, we avoid mis-classifying content. For example, earlier naive pipelines would wrap everything in `\includegraphics` images (even text or math), which is unacceptable [12]. Here, we explicitly mark semantic roles so we can treat text as text, math as math, etc., yielding truly editable LaTeX.

Step 2: Content Extraction per Segment

With the document segmented, we extract each component's content in a specialized way for maximum fidelity:

2.1 Text Blocks (Paragraphs & Headings)

For **digital PDFs**, Docling likely already extracted the text exactly as in the PDF (preserving punctuation, etc.). We can use that text directly. If line breaks are irregular, we may reflow or join lines as needed (Docling's reading order logic usually handles multi-column or line ordering correctly).

For **scanned PDFs or low-quality text**, we have options: - Rely on Docling's OCR (Tesseract) output from the previous step. This is fast but may have errors with math or special symbols. - **Use Nougat for text as well:** Nougat is trained on scientific papers and can OCR full page images, often capturing text and math in context. We can run Nougat on each page image to get an *alternate* text extraction. Nougat might better handle things like italic variables, special characters, or fused words that Tesseract misses, thanks to its transformer-based approach.

Using Nougat on pages: We can either call it via CLI or integrate in Python. For example, to process a PDF to Mathpix-Markdown:

```
nougat input.pdf -o nougat_output/ --model 0.1.0-base --no-markdown
```

This command uses the **base model** (for higher accuracy, given the GPU can handle it) and outputs a `.mmd` file per page in `nougat_output/`. We use `--no-markdown` to get raw Nougat output (which already includes LaTeX for math) without additional Markdown postprocessing [13] [3]. The result is a text file with the page's content: paragraphs of text, inline math in `$...$`, display equations in `$$...$$`, and tables in a markdown-like format.

We can parse these `.mmd` files to extract the text for each region. Since we also have Docling's segmentation, one strategy is: - Use Docling's layout to identify text blocks, but fill their content from Nougat's output. For example, for each text block region (with known coordinates), find the corresponding text in the Nougat output by matching positions (this requires mapping coordinate spaces, but an approximate approach is to assume reading order alignment). - Alternatively, if the PDF is fully scanned, one might skip Docling's text OCR and directly use Nougat's whole-page text. In that case, we lose fine region demarcation, but we could attempt to infer paragraphs from the Nougat output structure.

In practice, **a hybrid approach works best**: - Use Docling's text extraction for straightforward text (especially in digital PDFs where it's exact). - Use Nougat as a cross-check or for complex segments (math-heavy lines or unusual fonts). - Merge or choose the best result for each segment. This can be manual or rule-based (for instance, if Docling (Tesseract) output contains `?` or obvious errors, replace that segment with Nougat's text).

Before adding text into LaTeX, consider **cleaning and normalization**: - Fix common OCR quirks: e.g., replace hyphenation artifacts, ensure quotation marks are proper LaTeX quotes, etc. - Escape any LaTeX special characters that appear in plain text (%, $, _, &, #, {, } need escaping unless already properly formatted). *Note:* Ampersands `&` in text are especially problematic if they end up in LaTeX tables or math mode by mistake [14]. We'll handle these during assembly or in a validation pass.

2.2 Mathematical Equations

Math formulas are critical to capture precisely. We avoid any rasterization (images) for math. Our pipeline uses specialized OCR on each formula region: - **Docling's approach:** Docling might identify formula regions and (if `do_formula_enrichment=True`) attempt to preserve LaTeX for them. However, if the PDF was scanned or if formula characters are not literally available, this won't yield LaTeX. We thus explicitly OCR formulas. - **Using pix2tex:** For each formula (especially display equations), we crop the formula area from the page image and run it through pix2tex's model to get LaTeX code [6]. With our GPU, pix2tex is quite fast and accurate on clear scans or rendered formulas.

For example, using pix2tex in Python:

```
from PIL import Image
from pix2tex.cli import LatexOCR

latexocr_model = LatexOCR()  # loads model into memory (uses GPU if available)
formula_img = Image.open("formula_region.png")
latex_code = latexocr_model(formula_img)
print("Detected LaTeX:", latex_code)
```

[5]

If we prefer CLI, we can use:

```
pix2tex formula_region.png
```

which will output LaTeX for the given image in the console [15].

Crop extraction: To get `formula_region.png`, we use the coordinates from Docling's layout. For example, using PyMuPDF:

```
import fitz  # PyMuPDF
doc_pdf = fitz.open("input.pdf")
page = doc_pdf[page_index]
# Assuming we have a bounding box from Docling for the formula (in PDF
coordinates):
rect = fitz.Rect(x0, y0, x1, y1)
pix = page.get_pixmap(matrix=fitz.Matrix(2, 2), clip=rect)
# 2x zoom for higher DPI
pix.save("formula_region.png")
```

We double the resolution (Matrix(2,2)) to ensure the formula image is high DPI for OCR accuracy. This image is then fed to pix2tex.

Inline math: For math within text lines (e.g., $E=mc^2$ in a sentence), we have two possibilities: - If Docling did not separate them, Nougat's output likely already includes the LaTeX in-line (since Nougat outputs inline math in `$...$`). We can use that directly. - If needed, we could detect inline math by patterns (for instance, Docling might mark them or we might just trust Nougat for inline cases rather than cropping each one).

Using Nougat's whole-page output for math is convenient because it tends to correctly delineate inline vs display math. For display equations, we still prefer pix2tex on the image for highest accuracy, then compare with Nougat's LaTeX output (which we have in the `.mmd`). In many cases, Nougat's guess and pix2tex's output will match; if they differ, we can choose the one that compiles correctly or looks more plausible (this can be a simple heuristic or manual check).

Tables: If the PDF has math in tables, handle those similarly: each cell's text or formula can be processed through the above methods.

2.3 Tables

Docling's layout parser often can identify table structures, including cell text. It might output tables as Markdown tables or a CSV-like structure in its JSON. For example, Docling can preserve tables' rows and columns in the `DoclingDocument`. We can retrieve that and then convert to a LaTeX table (`tabular` environment or the higher-level `tabularx`, etc.). The **booktabs** package is recommended for professional tables [16], so we will use it in LaTeX.

If Docling's output for a table is insufficient (for instance, OCR errors in cells or merged cells issues), we have alternatives: - Use **Nougat**: Its `.mmd` might contain a table in Markdown format (with `|` delimiters) [3]. We can parse that and map it to LaTeX. - Use a dedicated table OCR like Microsoft's Table Transformer (if open-source model available on Hugging Face) to refine structure. This is optional; often Docling+Nougat suffice.

For each table: 1. Extract the table's text content for each cell (from Docling's structured result or Nougat's markdown). 2. Determine the column alignment or any spanning cells. (If complex, manual adjustment might be needed.) 3. Generate LaTeX using `\begin{tabular}` or `\begin{tabularx}`. Include `\toprule`, `\midrule`, `\bottomrule` if using `booktabs` for nicer formatting [16]. 4. Escape any `&` or `%` in cell text. Replace them with `\&` etc., because in LaTeX `&` is a column separator.

2.4 Figures and Images

Any non-text content like diagrams, charts, or photos in the PDF should be extracted as image files and referenced in LaTeX: - Use PyMuPDF or Poppler to extract the image at original resolution (or higher if needed). Docling's layout gives the region; sometimes the PDF itself contains the image data which PyMuPDF can extract directly via `page.get_images()`. - Save each figure image (e.g., `figure1.png`, `figure2.jpg`). - In the LaTeX, we will use the `graphicx` package to include these: e.g.

```
\begin{figure}[ht]
  \centering
  \includegraphics[width=0.8\textwidth]{figure1.png}
```

```
  \caption{Caption text from PDF.}
\end{figure}
```

If Docling or Nougat picked up a caption (likely as a text block near the image), use that as the `\caption`.

We ensure figures are *truly figures* (not misclassified content). The layout model typically distinguishes images by detecting non-text regions. This avoids the pitfall of accidentally treating a formula or a decorative shape as an image figure.

Step 3: LaTeX Assembly

With all pieces (text, equations as LaTeX, tables, figures) extracted, we now assemble the **LaTeX source document**. This step involves choosing a document structure and injecting content in the right places:

3.1 Document Preamble: Start the `.tex` file with a suitable document class and usepackages:

```
\documentclass[12pt]{book}   % or report, for a textbook-like style [17]
\usepackage[margin=1in]{geometry}
\usepackage{amsmath, amssymb, amsfonts, amsthm}
\usepackage{graphicx,xcolor}
\usepackage{booktabs, siunitx}   % for high-quality tables
\usepackage{tcolorbox}           % for any framed environments (optional)
\begin{document}
```

We prefer a book/report class with 12pt for a high-quality look [17], and include: - **AMS math packages** for robust math rendering (`amsmath` etc.). - **graphicx** for including images. - **booktabs** (for tables) and `siunitx` if numerical alignment is needed. - Possibly `enumitem` if we need custom list formatting for multi-part problems, and `tcolorbox` if we plan to style certain blocks like "Question" environments (as suggested in the blueprint for semantic blocks) [18].

3.2 Mapping segments to LaTeX: Iterate through the structured content (page by page, or logically): - For **headings**, use `\section{...}`, `\subsection{...}` as appropriate. If Docling identifies a top-level title or section headers, map those to LaTeX sections. - For **paragraph text**, simply place the text in the document. LaTeX will handle line breaking. Ensure any inline math is wrapped in `$...$` (Nougat usually has done so). Double-check that percentages, underscores, etc., are escaped or wrapped in `\text{}` as needed. - For **display equations** (from pix2tex/Nougat): If the LaTeX code is standalone, put it in an equation environment. We can use an unnumbered environment like:

```
\[
  % equation code here
\]
```

or if numbering is needed, `\begin{equation} ...\end{equation}`. If the original had alignment (say a multi-line derivation), and if we detected that (Nougat might output an `align*` environment or we can

infer if there are multiple aligned equals signs), use `\begin{align*} ... \end{align*}` for better formatting [19] . - For **inline math**, ensure it's within `$ ... $`. Nougat does this. If any inline math was missed and is just plain text (e.g., "max (a,b)" that should be math `$\max(a,b)$`), manually or programmatically adjust based on context. - For **tables**, format as discussed:

```
\begin{table}[h]
\centering
\begin{tabular}{cc|c}
  \toprule
  Column1 & Column2 & Column3 \\
  \midrule
  1 & 2 & $x^2$ \\
  3 & 4 & $y^2$ \\
  \bottomrule
\end{tabular}
\caption{...}
\end{table}
```

Use the content extracted for each cell. If the table spans the page width, consider `tabularx` environment with `\Xhline` for rules (or keep it simple with normal tabular and adjust column widths). - For **figures**, as shown, use `figure` environment with `\includegraphics`. If two images were side by side in the original, consider a `minipage` approach inside a single figure to replicate layout [20] . - For any **special text** (like code listings or footnotes), include appropriate packages (`listings` or use `verbatim` environment for code, etc.). Docling can detect code blocks as well, which we could then wrap in a `\begin{verbatim}` or minted environment (if we include minted and Pygments). This depends on the content type.

As we assemble, we maintain the logical reading order and group related content. For example, a "Question" in a textbook might consist of a text block followed by sub-parts and equations. We could wrap that in a custom environment:

```
\begin{question}
Text of the question here.

\begin{align*}
  ... % math derivation
\end{align*}
\end{question}
```

if we have defined `question` via tcolorbox [18] . However, implementing custom environments is optional – a simpler approach is to use sections or bold headings to denote these if styling isn't critical.

3.3 Closing document:

```
\end{document}
```

Once all content is placed.

Now we have a complete LaTeX source representing the PDF's content structure. The emphasis is on *semantic fidelity*: text is text, math is in math mode, tables are structured, and figures are included as graphics. This ensures the output is *searchable, editable, and accessible*, unlike a direct PDF or an image-based approach [12].

Step 4: Performance Optimizations

The given hardware (RTX 5090 32 GiB VRAM, 32 GiB RAM, 12-core CPU) allows us to maximize throughput and quality. Here are optimization tips for each stage:

- **Use GPU-accelerated models whenever possible:** We configured Docling's `pipeline="vlm"` to utilize a visual layout model (like *GraniteDocling*) which offloads layout analysis and even some OCR to the GPU. Nougat and pix2tex are both GPU-based by default (using PyTorch). This dramatically speeds up processing of pages compared to CPU-based OCR. For example, Nougat's base model can process a page in a few seconds on a modern GPU, as opposed to tens of seconds or more via CPU OCR for heavy math pages.

- **Batch processing:** Nougat CLI/API supports batching pages. We can adjust `--batchsize` for Nougat to process multiple pages in parallel [21], filling up the GPU. With 32 GiB VRAM, one can often use batch sizes of 2–4 pages (depending on page complexity and model size) to speed up processing of multi-page PDFs. Similarly, pix2tex's model can be applied in parallel to multiple formula images if we batch them through its PyTorch model (this might require minor code adaptation, as the `LatexOCR()` interface likely processes one image at a time, but one could loop and use asynchronous GPU calls).

- **CPU parallelism:** While the GPU handles heavy OCR tasks, utilize the 12-core CPU for parallel tasks:

- PDF image extraction (using PyMuPDF) can be done in parallel per page.
- If using Tesseract via Docling, note that Tesseract by default can use multiple cores for a single page OCR (it's multi-threaded). You can also process separate pages concurrently using Python threading or multiprocessing, since each page OCR is independent.
- If a PDF has many formulas, cropping all formula images can be parallelized, then sending them through pix2tex sequentially or in smaller batches.

- Use asynchronous I/O or multiprocessing to overlap I/O (reading/writing images, writing LaTeX file) with computation.

- **Memory considerations:** 32 GiB GPU memory is plenty for these models, but be mindful of loading multiple models:

- The Nougat base model, pix2tex model, and potentially a layout model (if not the same as Nougat) will all occupy VRAM. It's best to load one at a time or ensure the GPU can accommodate them concurrently. For instance, run layout analysis (Docling VLM) first (it might use a DETR or LayoutLM model), then unload it (if using Python, delete the model and clear CUDA cache) before running Nougat, etc., if memory is a concern.

- Alternatively, since Nougat itself performs layout+OCR, one could use Nougat for text/math and not use Docling's VLM on extremely large pages to save memory. But Docling's structure is very useful, so we keep it and manage memory by sequential processing.

- **Model size vs accuracy:** Nougat offers a smaller model (`0.1.0-small`) that runs faster and uses less VRAM, but for highest fidelity we choose the base model [3]. Given our 5090, we can comfortably run the base. The same goes for any alternative OCR (e.g., using TrOCR large models from HuggingFace could be another option if needed; those also would fit in 32 GiB).

- **Precision and performance:** Nougat uses bfloat16 by default for GPU inference, which is optimal on modern GPUs [22]. If you encounter any issues with bfloat16 (on older GPUs for example), you can use `--full-precision` to use FP32 at the cost of speed [22]. For our GPU, mixed precision is fine and speeds up inference.

- **Caching and reuse:** If processing many PDFs, keep the models loaded in memory across runs. For example, if writing a script, initialize Nougat and pix2tex models once, then loop over pages/documents. This avoids re-loading weights repeatedly.

- **Table processing:** For large tables, pure Python processing might become slow. Using NumPy/pandas to manipulate table data before LaTeX formatting can help if doing complex transforms. This is minor, but worth noting if there are hundreds of tables.

In summary, exploit the GPU for the heavy tasks (layout analysis, text and math OCR) and use the CPU's cores to handle parallelization and supporting tasks. This hardware can likely convert even lengthy textbooks relatively quickly with the above pipeline.

Step 5: Debugging & Quality Assurance

After assembling the LaTeX, it's crucial to validate that it compiles and that the content matches the source PDF. Here are strategies to debug and ensure quality:

- **Incremental compilation:** Use `latexmk` to compile the generated LaTeX document:

  ```
  latexmk -pdf -interaction=nonstopmode output.tex
  ```

  This will produce `output.pdf` if compilation succeeds. The `nonstopmode` flag ensures the compile doesn't pause on errors (errors will be logged instead).

- **Check for LaTeX errors:** Common issues and their fixes:

- **Misplaced alignment (&) errors:** If you see *"Misplaced alignment tab character &"*, it means an `&` was present in text or math where LaTeX interpreted it as alignment. For instance, "Domain & Range" in a text block inside math mode would trigger this [14]. The fix is to ensure such text is not in math mode or escape the `&` as `\&` in normal text. Our pipeline avoids putting text in math mode; such an error likely indicates a misclassification (e.g., treating an entire line as equation when it was text). The remedy is to adjust the segmentation or just escape the ampersand in the LaTeX.
- **Undefined control sequences:** For example `\inR` vs `\in \mathbb{R}` [23]. OCR might read a symbol poorly (missing a space or backslash). If `\inR` appears, LaTeX will complain that `\inR` is undefined. In this case, search and replace such patterns (`\inR` -> `\in \mathbb{R}`) to correct the notation. Developing a small library of these fixes from experience is helpful.
- **Missing packages for certain commands:** If the PDF had, say, script fonts or special symbols, ensure the corresponding LaTeX packages are included (e.g., `\mathcal` is covered by ams, but something like \mathscr might need `\usepackage{mathrsfs}`). If you see undefined commands, identify which package provides them or define them manually.

- **Overfull/Underfull hboxes:** These warnings indicate formatting issues (line breaks, etc.). They don't stop compilation, but if severe, you might adjust by adding `\sloppy` or reformatting the text slightly. For fidelity, you might not worry too much unless it affects appearance dramatically.

- **Automated log parsing:** For a systematic approach, use a tool like `texoutparse` or similar to parse the LaTeX log file for errors and warnings [24]. You can write a script to:

- Compile the LaTeX.
- Parse the `.log` for errors/warnings.
- If errors found, map them to segments and attempt fixes. For example, if the log indicates an issue in a certain line, you can trace which PDF segment that corresponds to (maybe store mapping during assembly).

- Apply known fixes (like the `&` or `\inR` as above), then recompile.

- **Visual comparison:** Open the original PDF and the generated LaTeX PDF side by side. Check that:

- All text content is present and in order.
- Equations look identical (no symbols lost or weirdly interpreted).
- Figures are placed and captions match.
- Tables have the same data and roughly similar layout.

If something is off (e.g., an equation rendered incorrectly), revisit the extraction for that element. This is where our optional fallback can come in: for example, if Nougat's output for an equation was used and looks wrong, try pix2tex, or vice versa, then update the LaTeX.

- **No unintended image fallbacks:** Ensure that the LaTeX source does **not** contain any `\includegraphics{...}` for content that should be text or math. All math should be actual LaTeX. This pipeline's design avoids that earlier "image embedding" fallacy [12], but double-check to be safe. If you find any, it means an OCR failed and an image slipped through – replace it by running an OCR on that image or typing it out.

- **Inline math vs text:** Sometimes OCR might confuse a math expression as normal text or vice versa. If you see things like `$DNA$` or other words in math mode that shouldn't be, or missing dollars for things that are variables, adjust those. This might require consulting the context or domain knowledge (e.g., if the PDF is scientific, single letters might be variables and should be italic/math mode). In ambiguous cases, accept what the original formatting suggests.

By iterating through compile -> inspect -> refine, you can gradually reach a "textbook-quality" output. The self-healing loop described in the internal design blueprint involves automating this refinement with an AI agent [24], but since we avoid cloud or external APIs, you could optionally use a local LLM (if available) to suggest fixes. Otherwise, manual fixes guided by logs will do.

Step 6: Optional Enhancements

Finally, once the core pipeline is working, we can consider enhancements to make it even more powerful and intelligent:

- **Local RAG (Retrieval-Augmented Generation) for Error Correction:** As an offline add-on, maintain a corpus of previous LaTeX documents or known correct formulas. If the OCR outputs a suspicious formula or text, the pipeline can perform a local similarity search. For example, if an equation OCR yields `E = m c^2` but in past documents the formula `E = mc^2` (without space) is common, the system can automatically correct formatting. Using libraries like *Haystack* or *LlamaIndex* (which Docling can integrate with [2] [25]), one can query a local vector index of texts for similar content. This is especially useful for recurring expressions or terminology: the system "recognizes" them from past data and can fix OCR inconsistencies (like variable naming, notation style).

Another use of RAG: If the PDF conversion is part of a Q&A system, after obtaining LaTeX, you could index it for semantic search or question-answering without needing the PDF again. This goes beyond conversion into knowledge management.

- **Multiple OCR engines for consensus:** Improve accuracy by running more than one OCR on the same content and comparing results. For instance, run both Nougat and Tesseract on body text: if they differ significantly for a line, flag it for review or choose the one more in line with a language model's expectation. Similarly for equations, you could run both Nougat and pix2tex; if one produces a LaTeX that fails to compile or looks invalid, prefer the other. This "ensemble" approach can boost reliability at the cost of performance.

- **Image-to-equation fallback & confidence checking:** pix2tex doesn't provide an explicit confidence score, but one can heuristically gauge output (e.g., very short outputs or presence of unusual tokens might indicate an error). If a formula conversion seems suspect, we can fallback to an alternative:

- If we primarily used pix2tex, try Nougat's formula output for that region (by isolating the formula in the page and reading Nougat's text).
- If we used Nougat and got `[MISSING_FORMULA]` or gibberish, definitely use pix2tex on that formula image.

- In extreme cases, even Tesseract with a math-aware font library could be tried, or asking a local GPT-4 model to parse the image (if one fine-tuned on LaTeX is available).

- **Structured metadata and labeling:** The pipeline could tag certain text as “definitions”, “theorems”, “proofs” if they follow known patterns, and wrap them in corresponding LaTeX environments (`\begin{theorem}...\end{theorem}` etc.). This moves toward a *semantic LaTeX* reproduction, useful in academic or textbook contexts. Achieving this might involve an NLP step on the text (which could be done with a local model or rules).

- **User feedback loop:** If the pipeline will be used regularly, implement a mechanism to correct errors once and propagate them. For example, if a particular unusual symbol was mis-OCRed, add a rule to fix it next time. Over time, the pipeline “learns” from its mistakes (even without an AI model, simple dictionaries or regex replacements can be powerful).

By combining these enhancements, we approach a robust system that not only converts PDFs to LaTeX offline but continuously improves and can leverage past knowledge.

In conclusion, this pipeline uses *Docling* for understanding document layout, *Nougat* and *pix2tex* for extracting text and equations with high accuracy, and carefully assembles the LaTeX using appropriate environments. The entire process is local, free, and accelerated by the available GPU, aligning with the requirement for a cloud-free, high-fidelity solution. By following the step-by-step guide and utilizing the provided tools and tips, one can convert complex PDFs (scanned or digital) into clean LaTeX ready for editing or publishing [1] [3].

Sources:

- Blecher et al., *Nougat: Neural Optical Understanding for Academic Documents* – Open-source PDF-to-markup model (Meta AI) [3] [22]
- Lukas Blecher, *pix2tex (LaTeX-OCR)* – Open-source equation OCR tool using ViT+Transformer [4] [5]
- IBM Docling – Open-source document parser for advanced PDF layout understanding (tables, formulas, code) [1] [2]
- “Analysis & Architectural Blueprint: PDF → LaTeX Conversion” – Internal design document outlining the multi-phase approach (layout analysis, specialized OCR agents, template-based generation, validation loop) [10] [6]
- Docling Documentation – Installation and OCR integration (Tesseract on Fedora, etc.) [7] [9]
- LaTeXify Project Logs – Discussion of common LaTeX conversion pitfalls (e.g., `&` alignment issues, macros like `\inR`) [14]

---

[1] [2] [25] GitHub - docling-project/docling: Get your documents ready for gen AI
https://github.com/docling-project/docling

[3] [13] [21] [22] GitHub - facebookresearch/nougat: Implementation of Nougat Neural Optical Understanding for Academic Documents
https://github.com/facebookresearch/nougat

[4] [5] [15] GitHub - lukas-blecher/LaTeX-OCR: pix2tex: Using a ViT to convert images of equations into LaTeX code.
https://github.com/lukas-blecher/LaTeX-OCR

6 10 12 14 16 17 18 19 20 23 24 BLUEPRINT.md
https://github.com/vladdiethecoder/LaTeXify/blob/6245aaea0de1c866e2db2aad03416f7119a76559/docs/BLUEPRINT.md

7 8 9 Installation - Docling
https://docling-project.github.io/docling/getting_started/installation/

11 Inline math: support for latex style formatting · Issue #2352 · docling-project/docling · GitHub
https://github.com/docling-project/docling/issues/2352