ChatGPT

Evaluation of PDF-DocX to JSON (Docling) Pipeline v1

Accuracy and Generality of Extraction

The PDF/DOCX to JSON pipeline demonstrates **high accuracy** in capturing form data, reportedly exceeding 95% field capture on typical dental intake forms [1]. The text extraction step uses robust libraries (PyMuPDF for PDFs and python-docx for Word files) to reliably get all textual content. In PDFs with an embedded text layer, nearly all printed text is captured. For scanned or image-only PDFs, the tool automatically falls back to OCR (Tesseract) to retrieve text, ensuring no form is skipped due to missing text [2] [3]. This guarantees that both digital and scanned forms have their content extracted for parsing.

Critically, the parsing logic is **form-agnostic** and not hardcoded to any specific form. It relies on generic regex patterns and a template dictionary to identify fields, rather than position or form-specific rules [4]. This general design allows it to handle a variety of dental form layouts and question phrasings. For example, the parser recognizes common field labels (Name, Date of Birth, etc.), yes/no questions, checkboxes, and text inputs using patterns rather than expecting exact wording. It also normalizes different phrasings to standard keys (e.g. "DOB" -> `date_of_birth`) via a comprehensive dictionary of 200+ fields. Overall, the extraction stage accurately retrieves text from both PDFs and DOCX files, and the parsing stage generalizes well across different form content.

Robustness to Different Form Layouts

The pipeline shows strong **robustness to various form layouts**, meaning it can parse information correctly from forms with different structures and formatting. Several enhancements have specifically improved its ability to handle edge-case layouts without custom code for each form. Notably, the latest version added support for **multi-part field labels**, **multi-column checkbox grids**, and **inline checkboxes**, which are all common in dental forms [5]. For instance, a line like *"Phone: Mobile __ Home __ Work __"* is now automatically split into separate fields (`phone_mobile`, `phone_home`, `phone_work`) instead of being lumped together [5]. Likewise, multi-column medical history checklists are detected and combined into one question with multiple options, even if spread across rows and columns. Section headers in all-caps or with varying phrasing (e.g. *"Dental History"* vs *"Dental Information"*) are recognized and normalized to consistent section names. Crucially, **no form-specific hardcoding** is used – all these cases are handled through pattern matching and inference rules [4]. This means the parser is flexible and can adapt to new forms that follow standard conventions, without needing per-form tweaks. The result is a form-agnostic solution where adding support for a new layout usually means adding a new regex or alias, not rewriting core logic. In summary, the pipeline handles different layouts gracefully, and the maintainers have proactively covered many edge cases (multi-field lines, grids, conditional questions, etc.) in the current version.

Performance and Resource Usage

Performance of the conversion process is efficient for typical use. Converting a single form from text to JSON is usually completed in under a second, thanks to the use of optimized libraries and in-memory processing [6]. The pipeline avoids network calls entirely (no external OCR or APIs), so performance is bounded only by local CPU and I/O speeds [7]. Extracting text with PyMuPDF is very fast for PDFs with text layers, and even OCR is done selectively (only when needed) to save time. The implementation now automatically detects scanned PDFs and triggers OCR, so users don't pay the OCR cost unless absolutely necessary [2]. The OCR itself uses a moderate 300 DPI image resolution per page for accuracy, which is a reasonable trade-off between speed and text clarity.

For batch processing, the tool supports **parallel execution** across multiple CPU cores to improve throughput [8] [9]. Both extraction (`docling_extract.py`) and conversion (`docling_text_to_modento.py`) accept a `--jobs` parameter to run N files in parallel, which can significantly speed up processing dozens of forms on multi-core machines [10] [11]. The conversion stage also received a performance patch to cache the template dictionary in memory so that each worker process loads it only once [12] [13]. This reduces redundant disk I/O and JSON parsing when processing many files concurrently. Memory usage is modest – mainly storing text and JSON in Python – though OCR on very large pages can temporarily consume more memory for image buffers. In practice, typical forms (a few pages) have caused no issues.

Overall, resource usage is **well-managed**. The pipeline uses local CPU for parsing and OCR, avoiding external service costs. It cleans up file handles after reading PDFs and doesn't leave large objects lingering. There is minimal overhead beyond the core tasks of text extraction and regex parsing. The maintainers have shown attention to optimization (e.g., caching and parallelism), so the tool scales reasonably well to high volumes of forms.

Test Coverage and Documentation Quality

The project shows a strong commitment to **testing and documentation**, which adds to its reliability. There is a comprehensive automated test suite with **over 80 tests** covering unit tests for parsing functions and end-to-end integration tests for the full PDF→JSON pipeline [14]. These tests cover critical components like text preprocessing (line wrapping fixes, character normalization), field parsing logic, template matching (including alias and fuzzy matching), and even full conversions of sample forms (including OCR scenarios and parallel processing) [15] [16]. Such thorough test coverage greatly reduces the chance of regressions and ensures new changes don't break existing functionality. The tests use representative snippets rather than specific form data, which aligns with the form-agnostic philosophy [17]. Additionally, the project maintainers have included integration tests in recent patches, meaning the entire pipeline is validated on example input-output pairs automatically [18]. The presence of these tests and their breadth indicates a high level of confidence in the code's correctness across many scenarios.

The **documentation quality** is excellent. The repository provides multiple well-written guides, including a detailed README with setup and usage instructions, an **Architecture.md** explaining the system design and key algorithms, a quick reference guide for common tasks, and even an ACTIONABLE_ITEMS.md describing planned improvements. The README clearly outlines how the pipeline works and its features (e.g., text extraction methods, intelligent parsing, output format) [19] [20]. It also enumerates supported form types

and known limitations in a very transparent way [21] [22]. The architecture document delves into each parsing step and the rationale for design decisions (like using regexes over ML) [23]. This level of documentation is rarely seen in similar projects and greatly aids understanding and maintaining the code. In summary, both testing and documentation appear to be **first-class priorities** in this project, which is a strong indicator of production readiness and developer diligence.

Overall Grade

Grade: A. The repository demonstrates outstanding performance across all evaluated aspects. It accurately extracts and converts dental forms with a general solution, handles varied layouts robustly, and is efficient in operation. The thorough test suite and rich documentation further solidify its quality. The maintainers have been proactive in addressing previous feedback (implementing parallel processing, OCR auto-detection, integration tests, etc.), bringing the project to a very high standard. Minor areas for improvement remain (mostly around maintainability and a few edge cases), but none of these significantly detract from the overall excellence. This latest version clearly deserves an **A grade**, reflecting a mature and reliable solution for PDF/DOCX form conversion.

Identified Issues and Patch Suggestions

Despite the strong performance, a few improvement opportunities were noted. Each issue is outlined below with a targeted patch suggestion. All recommendations preserve the form-agnostic approach (no hard-coding to specific forms).

1. **Large Core Module (Maintainability)**
   File/Section: `docling_text_to_modento/core.py` (general parsing logic)
   Issue: The core parsing script is still very large (~4000 lines) and only partially modularized. Many helper functions and post-processing routines remain in this monolithic file [24], which can make it hard to navigate and maintain. Although several modules (e.g., `text_preprocessing.py`, `grid_parser.py`) were introduced, the refactor is not complete. This affects **maintainability** and **developer agility** – future enhancements or bug fixes require working in a huge file, increasing the chance of errors or overlooking interactions.
   Suggested Patch: Continue splitting `core.py` into logical modules:
2. Move field parsing and detection functions into `modules/question_parser.py` (e.g., functions that identify field labels, options, yes/no patterns).
3. Move post-processing functions into a new `modules/postprocessing.py` (e.g., merging duplicate fields, inferring sections).

4. Keep only high-level orchestration in `core.py` (reading input, calling module functions, writing output).
   For example, all functions named `postprocess_*` could be cut from `core.py` and pasted into `postprocessing.py`, then imported. Similarly, section detection logic (`is_heading`, etc.) can reside in `text_preprocessing.py`. Update imports in `core.py` accordingly. Pseudocode for refactoring:

```
# In modules/postprocessing.py
def postprocess_consolidate_duplicates(payload, dbg=None):
    ...  # (move implementation from core.py)

def postprocess_infer_sections(payload, dbg=None):
    ...  # (move implementation from core.py)
```

And in `core.py`:

```
from .modules import postprocessing
...
payload = postprocessing.postprocess_consolidate_duplicates(payload,
dbg=dbg)
payload = postprocessing.postprocess_infer_sections(payload, dbg=dbg)
```

Each move should be accompanied by running the test suite to ensure nothing breaks.
Benefit: Further modularization will **improve code readability and maintainability**. Developers can focus on one aspect of parsing at a time, and testing individual modules becomes easier. It reduces cognitive load by shrinking `core.py` to a manageable size (target <1000 lines). In the long run, this makes adding new features or patterns safer and faster, since the code is organized by functionality. (This is a refactoring improvement; it doesn't change runtime behavior, but greatly helps ongoing development.)

5. **Capture Filled PDF Form Fields**
   File/Section: `docling_extract.py` in function `extract_text_from_pdf()`
   Issue: When PDFs are **fillable forms** (with AcroForm fields that a user can type into), the current extraction relies on `page.get_text()` which may not capture the values in form fields if they aren't flattened. If a patient has filled out a PDF form digitally, there's a risk that the typed answers (which are PDF form field values) won't appear in the extracted text, thus missing data. The tool encourages using fillable PDFs for consistency [25], so it should ensure the filled values are extracted.
   Suggested Patch: After extracting normal text from each PDF page, explicitly retrieve values of PDF form fields (widgets) and append them to the text if not already present. PyMuPDF allows iteration over PDF widgets. For example:

```
doc = fitz.open(file_path)
text_parts = []
for page in doc:
    page_text = page.get_text("text")
    text_parts.append(page_text)
    # New: capture form field values on this page
    if page.widgets():
        for widget in page.widgets():
            val = widget.field_value
            name = widget.field_name or "Field"
            if val:  # if a value is present
```

```
                text_parts.append(f"{name}: {val}")
doc.close()
text = "\n".join(text_parts)
```

This will add lines like "`FieldName: FilledValue`" to the extracted text, which the parser can then pick up as if they were part of the form. (If the field value already appears in `page_text` due to an appearance stream, this will harmlessly duplicate it; de-duplication can be done if necessary by tracking seen lines.)
Benefit: Ensures **no loss of data** for digitally filled PDFs. All patient-provided answers will be captured, improving accuracy for cases where forms are completed electronically. This makes the extraction truly comprehensive for modern PDF forms. It aligns with the goal of accuracy and prevents silent omissions of key information.

6. **Graceful Handling of Non-Extractable PDFs without OCR**
File/Section: `docling_extract.py` in `process_one()` or `extract_text_from_pdf()`
Issue: If a PDF has no text layer and OCR is **not available** (e.g., Tesseract not installed) or auto-OCR is turned off, the extractor currently writes out a placeholder text file containing an error message (e.g., `"[NO TEXT LAYER] ... OCR is not available."`) [3]. While this notifies the user, the pipeline may still attempt to convert that `.txt` file in stage 2, resulting in an empty or meaningless JSON. Essentially, a form that could not be processed ends up generating a dummy output. This is not harmful, but it could be confusing or lead to empty JSONs in the output directory.
Suggested Patch: Skip JSON conversion for files that were not actually extracted. Specifically, in `process_one()` of `docling_extract.py`, if extraction fails or yields the special "[NO TEXT LAYER]" message, do not write a `.txt` (or mark it to be skipped). For example:

```
text = extract_text_from_pdf(file_path, ...)
if text.startswith("[NO TEXT LAYER]"):
    print(f"[!] Skipping {file_path.name} – no text and OCR unavailable",
file=sys.stderr)
    return  # do not write an output .txt
```

Additionally, the conversion script (`docling_text_to_modento.py`) could detect if a text file contains no actual fields (perhaps by checking if the first line starts with `[NO TEXT LAYER]`) and then skip generating a JSON for it, logging a warning instead. This check can prevent creating an empty JSON.
Benefit: Provides a more **graceful handling** of completely unextractable documents. Instead of producing empty outputs, the pipeline will clearly skip those files, and the user can see the warning to install OCR or handle that form separately. This improves the user experience by avoiding ambiguous results (an empty JSON) and ensuring that every JSON produced corresponds to a real successfully processed form. It essentially fails fast and loud for the small number of cases that truly cannot be processed with the available tools, prompting user intervention.

Each of the above patches is designed to enhance the system without sacrificing its generality or flexibility. Implementing these would further polish an already high-quality project: making the codebase easier to

maintain, covering a niche data-extraction gap, and handling edge cases more cleanly. None of these changes require hardcoding for specific forms; they are generic improvements aligned with the project's architecture and goals. With these refinements, the PDF-to-JSON Docling pipeline would move even closer to a rock-solid, production-ready state.

Sources:

- Project README – *Features, usage, and limitations* [19] [22]
- Architecture Documentation – *Design of parsing logic and decisions* [23] [26]
- Evaluation Analysis – *Prior assessment notes on accuracy, robustness, and performance* [1] [24]
- Test Suite Documentation – *Scope of tests and coverage* [15] [14]
- Source Code Excerpts – *Implementation details for OCR, parallelism, etc.* [2] [13]

---

[1] [6] [12] [24] EVALUATION_FEEDBACK_ANALYSIS.md
https://github.com/rontavious999/pdf-docx-to-json-docling-v1/blob/83d4b5a07842c023e235ba70b9cbf8c2c339f1fc/EVALUATION_FEEDBACK_ANALYSIS.md

[2] [3] [9] [11] docling_extract.py
https://github.com/rontavious999/pdf-docx-to-json-docling-v1/blob/83d4b5a07842c023e235ba70b9cbf8c2c339f1fc/docling_extract.py

[4] [13] core.py
https://github.com/rontavious999/pdf-docx-to-json-docling-v1/blob/83d4b5a07842c023e235ba70b9cbf8c2c339f1fc/docling_text_to_modento/core.py

[5] [8] [10] [19] [20] [21] [22] [25] README.md
https://github.com/rontavious999/pdf-docx-to-json-docling-v1/blob/83d4b5a07842c023e235ba70b9cbf8c2c339f1fc/README.md

[7] [23] [26] ARCHITECTURE.md
https://github.com/rontavious999/pdf-docx-to-json-docling-v1/blob/83d4b5a07842c023e235ba70b9cbf8c2c339f1fc/ARCHITECTURE.md

[14] [15] [16] [17] [18] README.md
https://github.com/rontavious999/pdf-docx-to-json-docling-v1/blob/83d4b5a07842c023e235ba70b9cbf8c2c339f1fc/tests/README.md