ChatGPT

Best Libraries for Bilingual (Arabic/English) PDF Text Extraction

Extracting structured text from **complex bilingual PDFs** (e.g. statistical yearbooks with side-by-side Arabic and English content) is challenging. The ideal solution must handle multi-column layouts, preserve logical structure (paragraphs, headers, tables, footnotes), and properly support Arabic right-to-left (RTL) text (including correct character order, punctuation, and diacritics). Below we evaluate top open-source and commercial options against these criteria, and recommend the best choice for a Retrieval-Augmented Generation (RAG) pipeline.

Key Criteria for Extraction in RAG Pipelines

- **Layout Complexity:** Handling of multi-column pages, mixed-language pages, tables, headers/footers, etc., without mixing content out of order. The library should detect and separate columns or regions so text flows correctly.
- **Structure Preservation:** Output should retain meaningful divisions – e.g. paragraphs, list items, table cells – rather than a raw text jumble. Page-level alignment of parallel Arabic/English text is important so corresponding segments can be matched if needed.
- **Arabic Language Support:** Must correctly extract Arabic text in reading order (RTL) with proper letter shaping. Many PDF extractors give reversed or disjointed Arabic text, requiring fixes. Ideal solution should minimize this or provide ways to correct it.
- **Clean Output for Chunking:** The extracted text should be *clean* (minimal artifacts) and segmented in a way that downstream chunking (e.g. via ChunkWise or LangChain) can easily consume – e.g. one paragraph or table per chunk with semantic coherence.
- **Performance and Scale:** Since RAG pipelines may process thousands of pages, the library should be efficient in Python, with reasonable speed and memory usage. Support for parallel processing or incremental parsing is a plus.

With these requirements in mind, let's compare the leading libraries:

PDFPlumber (Open-Source)

Overview: pdfplumber is a Python library built on PDFMiner that provides detailed access to PDF content (text, fonts, layout coordinates) and includes helper functions for extracting text and tables. It's known for robust table extraction and gives fine-grained control over layout elements.

- **Layout & Structure:** By default, `pdfplumber.Page.extract_text()` will give a plain text string of the page. However, it may **mix multi-column text** if used naively (text from different columns can be concatenated in the wrong order [1]). To handle complex layouts, you can either manually crop pages into regions or leverage PDFMiner's layout analysis via `laparams`. In fact, pdfplumber can expose PDFMiner's higher-level objects (e.g. text boxes for each column) if you pass a `laparams` configuration [2]. This allows identifying separate text blocks (columns, headers, footnotes) rather than one continuous flow. Using such configuration or manual region cropping, pdfplumber **can preserve paragraphs and column structure**, but it requires extra work from the developer (no one-click multi-column magic). On the plus side, pdfplumber's ability to access coordinates of each character/line means you can implement custom logic to reassemble text in reading order (e.g. iterate text boxes left-to-right, top-to-bottom).

- **Tables & Figures:** pdfplumber shines for table extraction. Its `page.extract_table()` uses detected lines and whitespace to output tabular data [3]. This is useful for statistical yearbooks with many tables – you can get structured rows/columns directly. It does not reconstruct images or complex figures (no OCR built-in), focusing on text and lines.

- **Arabic RTL Support:** Raw output from pdfplumber (and PDFMiner) will have Arabic text **characters reversed and diacritics separated by spaces** in many cases [4]. This is because PDF encodes the text in a logical order that doesn't match visual RTL order. **Workarounds:** pdfplumber (since v0.7.4) added parameters `line_dir_render` and `char_dir_render` to assist with bidi rendering. For example, `page.extract_text(line_dir_render="ttb", char_dir_render="rtl")` can yield correctly ordered Arabic text in some cases without manual post-processing [5]. However, this approach may not work for all fonts or PDFs – it can be *fragile*, as one user noted [6]. The more general solution is to post-process with Arabic shaping libraries. A common pipeline is: use pdfplumber to extract the text *in logical order*, then apply python-bidi and arabic_reshaper to reorder characters and fix their presentation forms [7] [8]. This was shown to turn pdfplumber's initial output (which was backwards with extraneous spaces) into properly readable Arabic text [7] [9]. In short, pdfplumber **can** extract Arabic, but you must plan to run a bidi algorithm and normalization on the result to get usable text.

- **Performance:** Being pure Python (built atop PDFMiner), pdfplumber is not the fastest. It's faster than some older pure-Python tools but slower than libraries with C/C++ backends. In one benchmark, extracting a page of text took ~0.10 seconds with pdfplumber [10] – fast enough for many cases, but not as quick as MuPDF (which can be ~10x faster) [11]. It also has memory overhead proportional to PDF size. You can speed up large jobs by processing pages in parallel (since each page can be opened independently), but pdfplumber itself doesn't have built-in parallelism.

Use Case Fit: pdfplumber is a great choice if your PDF has lots of **tables or irregular layouts**, and you need fine control. It's battle-tested in data journalism and PDF data extraction tasks. However, to use it in a bilingual RAG pipeline, expect to add a post-processing step for Arabic text order/diacritics, and possibly custom layout handling for columns. If you need a quick, structured text dump (paragraphs in order) with minimal coding, pdfplumber will require some tuning – it doesn't automatically label sections or maintain bilingual alignment beyond basic page coordinates.

PyMuPDF (MuPDF) – via PyMuPDF or PyMuPDF4LLM (Open-Source)

Overview: PyMuPDF is a Python binding for the MuPDF library, a lightweight PDF and XPS rendering engine. It is extremely fast and can extract text, metadata, and images. Recent community additions (like `pymupdf4llm`) build on PyMuPDF to produce structured outputs (like Markdown).

- **Layout & Structure:** PyMuPDF by default provides several text extraction modes. For example, `page.get_text("text")` gives a continuous text stream per page, and `page.get_text("blocks")` returns a list of text blocks with their bounding boxes. Using the blocks output, you can distinguish different regions (columns, sidebars, etc.). In fact, MuPDF has an internal concept of text blocks that often correspond to paragraphs or columns. There is a known utility script for PyMuPDF called `multi_column.py` that leverages these blocks to **detect multiple columns** on a page [12] [13]. This script computes column boundary boxes and allows you to extract text from each column in sequence [14]. It even lets you ignore page footers by setting a margin (so repeated page footnotes or page numbers aren't intermingled with main text) [15]. Using such an approach, PyMuPDF can accurately preserve the reading order on multi-column layouts – e.g. first extract left column text, then right column – instead of mixing them. *Note:* the provided script currently assumes left-to-right text direction when sequencing columns [16], so a bit of adaptation might be needed for RTL content, but the concept (segmenting columns by position) still holds for Arabic-English pages (you'd just treat the Arabic column as a separate block).

- **Arabic RTL Support:** As with pdfplumber, raw text from PyMuPDF will not be RTL-correct. MuPDF doesn't perform bidirectional reordering or Arabic shaping on extraction – it yields characters in the internal storage order. One observed issue is with Arabic ligatures: e.g. the glyph for "ﻻ" (lam-alef) might be returned as two characters but not reversed with the rest of the word, causing jumbled output [17] [18]. The PyMuPDF maintainers have noted this and currently do **not attempt to resolve RTL ordering or ligature issues internally** (the issue was closed as "wontfix", expecting the user to handle it) [19] [20]. Therefore, similar to pdfplumber, you will need to run the extracted text through `bidi.algorithm.get_display()` and possibly `arabic_reshaper` to get proper Arabic sentences [21] [22]. This extra step is essentially mandatory for any open-source extractor (PyMuPDF, PDFMiner, etc.) when dealing with Arabic. The good news is that once you do this, PyMuPDF's output is otherwise clean (no weird spacing issues beyond the ligature splitting). In summary, PyMuPDF handles Arabic characters and Unicode fine, but **relies on you to reorder RTL text** and join forms.

- **Performance:** PyMuPDF is one of the fastest libraries available. It's backed by C/C++ code and optimized for speed. Community tests show it can be an order of magnitude faster than PDFMiner-based tools [11]. In one real-world comparison, a PyMuPDF-based tool (`pymupdf4llm`) extracted a document in **0.12–0.14 seconds** (per page) with structured output [23] [24] – extremely efficient. This makes PyMuPDF very suitable for large PDFs or high-throughput pipelines. It also has low memory overhead, as it can load and process page by page.

- **Structured Output:** By default, PyMuPDF gives raw text or simple block groupings. However, there are higher-level tools built on it. Notably, *pymupdf4llm* (a utility built for LLM/RAG use cases) can output Markdown with headings, lists, tables, etc., inferred from the PDF structure [25]. Users reported that PyMuPDF (with such utilities) produces **clean structured text** (with proper hierarchy) that's great for downstream use [26]. The only caution is that extremely complex layouts (many columns or non-standard flows) might still confuse it [27] – but with the column detection techniques mentioned, it remains one of the best in preserving layout.

Use Case Fit: PyMuPDF is arguably the **top open-source choice** when performance and layout accuracy are needed. It can handle multi-column bilingual pages by separating blocks, and it's efficient enough to parse thousands of pages. You will need to integrate Arabic text post-processing (bidi/reshaping), as with others. PyMuPDF doesn't inherently label content types (no semantic tagging of "Title" vs "Body" text), so the output will be raw text (or Markdown) broken into sections by layout – which is generally fine for chunking. Given its speed and flexibility, PyMuPDF (perhaps in combination with the `pymupdf4llm` or the multi-column script) hits a sweet spot for RAG pipelines [28]. Many practitioners choose it for building Q&A over PDFs due to this balance of **speed and quality** [29].

Unstructured (Open-Source)

Overview: *Unstructured* is a newer library focused on preparing documents for LLM consumption. It provides a high-level API to "partition" documents (PDFs, HTML, etc.) into semantic elements. Each element is labeled (e.g. Narrative text, Title, List item) and can be passed directly to an embedding or QA system. Under the hood, Unstructured uses a combination of techniques: it may use PDFMiner or image-based layout parsing (with OCR) depending on the chosen strategy (`hi_res`, `fast`, etc.).

- **Layout & Structure:** Unstructured's goal is to preserve semantic structure rather than exact visual layout. For example, calling `partition_pdf(...)` on a PDF returns a list of elements like Title, Section Header, Paragraph, List, Table, etc., each with the text content and metadata [30]. This is very useful in RAG pipelines – it gives you *meaningful chunks* (with boundaries that align to the document's logical structure) out-of-the-box [31]. It will also keep elements in the correct reading order. In a bilingual document, one would expect it to return interleaved Arabic and English elements as they appear on the page (likely by reading order). However, preserving the **alignment** between parallel texts is not explicitly guaranteed – it depends on whether the parsing groups them separately or as one element. Typically, if Arabic and English text are in separate columns or blocks, Unstructured will output them as separate elements (with perhaps spatial metadata like page number or coordinates that you could use to align if needed).

- **Arabic RTL Support:** This is a known weak point as of early 2025. By default, Unstructured (when using the default or hi_res strategy) can end up outputting Arabic text in reversed character order (left-to-right) [32]. A bug report from Feb 2025 shows that an Arabic phrase was extracted as garbled "Dlor wolleh" style output (i.e. reversed) [32]. This indicates that the library does not automatically apply bidi reordering. The maintainers will likely address this, but if not yet fixed, it means you might need to post-process Unstructured's Arabic outputs similar to the other libraries. The difference is that Unstructured splits the text into chunks already – so you'd need to run a bidi fix on each Arabic chunk individually. Moreover, if Unstructured's OCR-based mode is used (hi_res with language support), accuracy of Arabic text might depend on Tesseract's Arabic OCR, which can introduce recognition errors (though the bug example seemed to be text, not OCR). In short, **Arabic handling is currently a drawback** – you get nicely separated content, but the Arabic content may require reordering.

- **Complex Layouts:** Unstructured is designed to handle **headers, footers, and multi-column** flows by using a layout detection model. The "hi_res" strategy uses computer vision (layout parsing) to detect text blocks, which should naturally separate columns and ignore repetitive headers/footers. This means it can *automatically* drop those page numbers or running heads that you might otherwise filter manually. It also attempts to reconstruct tables as `Table` elements (with cells that you can further process). These capabilities are very promising for a complicated report or yearbook. The trade-off is speed: the hi_res (layout ML) approach is significantly slower and more resource-intensive than pure text extraction.

- **Performance:** Unstructured's convenience comes at a cost. Parsing a large PDF through Unstructured can be quite slow – one user reported ~3 hours for a 2000-page PDF using the open-source pipeline [33]. This is because of the heavier processing (possibly rendering pages for layout analysis or OCR). The *fast* mode (which uses PDFMiner) is quicker but then doesn't do as much intelligent structuring. In a measured benchmark on a single document, Unstructured took ~1.29 seconds for a small file, versus ~0.12s for PyMuPDF on the same content [26] [34]. This gap grows larger on bigger files. For RAG pipelines that need to ingest many PDFs, this can become a bottleneck. That said, you *can* parallelize Unstructured by splitting the PDF by pages or sections and processing in threads or processes, to mitigate some of the slowness.

Use Case Fit: Unstructured is **tailor-made for RAG** in terms of output format – you directly get chunks with labels, which is extremely convenient [35]. If it handled Arabic perfectly, it might be the top choice. However, given the current RTL issues and performance considerations, it may require additional engineering (bidi fixes, possibly using a faster strategy or parallelization) to use on large bilingual PDFs. It's a good choice if you value the semantic labeling and have moderate document sizes (or can preprocess overnight, etc.). Otherwise, a lower-level library like PyMuPDF might be preferable for raw speed, with your own chunking logic on top.

PDFMiner.six (Open-Source)

Overview: pdfminer.six is the maintained version of PDFMiner (a pure Python PDF parsing library). It is essentially the engine behind pdfplumber's text extraction. It provides very granular control – you can parse a PDF into a hierarchy of pages, boxes, lines, characters, etc.

- **Layout & Structure:** PDFMiner's biggest strength is its configurable layout analyzer. By tuning parameters (in `LAParams` such as char margin, line margin, word margin), you can influence how text is grouped into lines and boxes. It will identify `LTTextBox` objects which often correspond to columns or paragraph blocks. This means you *can* get structured output (e.g. two text boxes for two columns on a page) and even detect the reading order if you use the `order` attributes it provides. However, using PDFMiner directly is complex – you might end up writing a lot of code to assemble the final text. Most people instead use it via a wrapper (like pdfplumber, which gives simpler calls and some defaults). Notably, PDFMiner has a command-line tool `pdf2txt.py` that can output text or HTML; some have used it to get HTML with position info as a form of structured extraction. But directly for RAG, using PDFMiner means essentially building your own extractor.

- **Arabic RTL Support:** PDFMiner shares the same limitations discussed before – it will output Arabic in logical order (which appears backwards). An issue was raised about Arabic ligature ordering (lam-alef) and the verdict was that PDFMiner doesn't handle that scenario [36] [37]. In other words, **no built-in RTL support**. You would again apply external fixes. One workaround some users found is to use **Poppler's pdftotext** (a separate tool, see below) which gave better initial ordering and then normalize [38] [39] – but that's outside PDFMiner.

- **Performance:** Being pure Python, PDFMiner is the slowest of the bunch. It can struggle on very large PDFs in terms of speed and memory. It's also single-threaded. This makes it less ideal for high-throughput extraction unless absolutely necessary.

Use Case Fit: Typically, you wouldn't pick PDFMiner alone as "the" library for this task – it's more of a building block. Since pdfplumber encapsulates PDFMiner with a nicer API (and they are always used together), one could consider pdfplumber the practical interface to PDFMiner for most uses. If extremely fine-tuned control is needed (e.g. custom parsing of each page's objects), PDFMiner is there, but for a RAG pipeline it likely adds unnecessary complexity.

Adobe PDF Services Extract API (Commercial)

Overview: Adobe's PDF Extract API (part of Adobe Document Services) is a cloud-based service that uses Adobe's own PDF parsing and AI (Sensei) to extract content. It outputs a structured JSON containing text elements, tables, images, and their styles/positions [40] [41]. Adobe essentially leverages the same technology behind Acrobat's "Export PDF" features.

- **Accuracy & Structure:** The Adobe API is known for producing a very faithful extraction. It **retains structure** such as paragraphs (with proper line merging), headings, list elements, table structures, and even reading order in complex layouts. It's designed to handle things like multi-column pages and identify headers/footers (often tagging them as such so you can ignore repetitive headers). In many enterprise use-cases, it's considered a gold standard for difficult PDFs. For bilingual documents, the API would treat each text run with its language – you'd likely get each paragraph of Arabic and English with metadata indicating direction or at least the correct order on page. Page-level alignment is inherently preserved because the JSON includes page number references for each element.

- **Arabic Support:** One caveat: Adobe's solution historically had limited language optimization beyond English. Officially, as of some time ago, *"the API is currently optimized for English… content in other Latin languages should return good results"* [42]. There was even a statement that Acrobat couldn't extract Arabic properly a few years back [43]. This suggests that while the Extract API will **return something** for Arabic text, it might not apply specialized logic for RTL ordering or font decoding. If the PDF text is properly encoded in Unicode, the JSON likely contains the correct Unicode characters (possibly still in logical order rather than visual). If not, you may face similar issues albeit in a structured format. We don't have a public detailed evaluation of Adobe on Arabic yearbooks, but it's safe to say **Adobe's engine will not lose characters or mix columns**, but you might still need to reverse the Arabic strings. The advantage is you'd get each piece clearly demarcated (so you know what to reverse and where it belongs).

- **Tables and Figures:** Adobe's API is quite adept at tables – it provides table structures with rows and columns identified in the JSON. It also extracts images and figures (though for RAG text content, you might ignore those). If your yearbook has statistical tables, this API will likely capture them cleanly, possibly more accurately than open-source table extractors in edge cases.

- **Integration & Performance:** Being a cloud API, you send the PDF and get results asynchronously (typically within seconds for even large PDFs, but network latency and size can affect it). For very large documents, you might need to chunk or use their asynchronous jobs. In terms of throughput, it's not as instantaneous as a local library like PyMuPDF – but it can be reasonably fast for moderate documents. You also have to consider **cost** (there's a usage-based pricing) and the need to handle API credentials, etc. If your pipeline can call external services and budget isn't an issue, this is fine. If you need offline or unlimited processing, a self-hosted solution is preferable.

Use Case Fit: Adobe's Extract API is a top choice when **highest fidelity extraction** is required and you're willing to use a commercial service. It handles complex layouts and yields a richly structured output ideal for further processing. In a bilingual RAG scenario, it would give you cleanly separated English and Arabic text blocks (with location data), likely needing minimal cleanup aside from ensuring Arabic reads correctly. The trade-offs are reliance on an external API and the uncertain RTL nuance. If evaluating commercial options, Adobe is a frontrunner, but one should also consider **Google Cloud Document AI** and **AWS Textract** as alternatives: these services also parse PDFs into structured forms and support Arabic via OCR. For instance, Google's Document AI has a specialized layout understanding and can output JSON with text language detected (their OCR is strong in Arabic). Textract can identify text and tables in Arabic as well. Each of these has its own pricing and performance profile. Adobe's advantage is its deep PDF expertise; Google's might be better OCR in some cases. In any case, commercial APIs will generally handle the heavy lifting of structure and layout, at the expense of external dependency.

Other Noteworthy Tools

Before concluding, a quick mention of a few other tools and their relevance:

- **Poppler's `pdftotext` Utility (Open-Source):** This is a command-line tool (C++ library) that many consider a benchmark for text extraction. It often **handles Unicode and layout well**. In fact, one user found that `pdftotext` produced correct Arabic text order and nearly perfect output for an Arabic book (after applying Unicode normalization to fix some character forms) [38] [39]. You can invoke Poppler via Python (e.g. using the `subprocess` module or through wrappers like `pdftotext` pip package). It doesn't give structured output (just plain text or HTML), but if your priority is **correct bilingual text content**, Poppler is an excellent engine. It automatically keeps logical reading order in many cases and might require less post-processing for Arabic (aside from normalization of certain ligatures [44]). The downside is you'd still need to split the text into chunks and it won't label or separate sections beyond what the text layout implies.

- **PyPDF / PyPDF2 (Open-Source):** PyPDF2 (now just `pypdf`) is a pure-Python PDF library that can extract text. It's stable and lightweight, but not focused on layout – it will often return text in content stream order (which can be scrambled in multi-column layouts). Its maintainer has acknowledged limitations with RTL languages [45]. Use this only for simple PDFs; it's not suitable for complex structured extraction needed here.

- **pypdfium2 (Open-Source):** A Python binding for PDFium (the engine used in Chrome). It's extremely fast and reliably gets text, but like pypdf, it doesn't group or analyze layout deeply – you'd get lines of text that you must assemble. It's an option if performance is king and you will handle structure yourself. However, for bilingual content, you'd still face the bidi issue and need to determine column groupings manually.

- **Deep Learning-based Parsers:** Libraries/models like **LayoutLMv3**, **Donut**, or **Marker** (by Weavy) use AI to read PDF content and produce structured outputs (Marker can even produce markdown with layout preserved [46] [47]). These can be very powerful (Marker produced "stunning layout-perfect markdown" in one test [48]), but they are heavy (multi-hundred MB models) and slower (seconds per page) [47]. They might be overkill for most RAG pipelines and require GPU support for good speed. Unless you have very irregular layouts that rule-based extractors can't handle, these are generally not necessary.

- **OCR Solutions:** If the PDFs are scanned images (not actual text), you will need OCR. Tesseract with Arabic language data can be used (possibly via frameworks like `textract` or Google's Tika which can call OCR). Tesseract's Arabic OCR accuracy is decent but not perfect, and it will not preserve structure beyond line breaks. Commercial OCR (Google Vision, AWS Textract, Azure OCR) tends to be more accurate for Arabic and can detect columns. Since the question assumes "complex PDF like yearbooks" (likely digitally published), we focused on non-OCR extraction. But it's worth noting for completeness.

Recommendation and Trade-offs

Considering all the above, **the best overall library for this use-case** (complex bilingual PDF extraction in a RAG pipeline) is **PyMuPDF (MuPDF)**, used with appropriate add-ons or custom code for layout and Arabic handling. PyMuPDF offers:

- **Accurate layout parsing** (multi-column support via block detection) to keep Arabic and English text properly separated and ordered [13].
- **High performance** (can handle large documents efficiently, ~10x faster than PDFMiner-based tools [11], crucial for RAG scalability).
- **Output flexibility** – you can extract text as needed (plain text, Markdown, or block-by-block) which can then be fed into your chunking pipeline easily.
- A strong track record in real-world PDF analytics (widely used in QA systems over PDFs).

What about Arabic text quality? With PyMuPDF you will still need a one-time post-processing step: pass the extracted text through an Arabic reshaper and bidi algorithm. This is a minor addition and is the same requirement you'd have with most other libraries (open-source or not) [21] [22]. The benefit is that PyMuPDF gives you the text *with minimal corruption* – no missing characters – so the fix is reliably effective. By contrast, some alternatives like pdfplumber also require bidi fixes and can involve cleaning up diacritic spacing issues [4] [49], and they run slower.

Comparing to Unstructured: If your priority is out-of-the-box chunking and you don't mind a heavier pipeline, Unstructured is a close contender. It will save you effort in splitting the text and identifying sections, which is valuable for RAG. However, at the time of writing it still struggles with Arabic RTL output [32], meaning you might have to do nearly the same post-processing anyway. Its performance is also a bottleneck for very large docs (though for a typical yearbook of, say, 300 pages, it might be workable with some patience or parallelism). Unstructured could become the top choice if/when it fully addresses RTL and improves speed, but **for now PyMuPDF is more reliable and efficient**.

Comparing to Adobe API: Adobe's PDF Extract API is arguably the most powerful in terms of layout and structure fidelity. If maximum accuracy is needed and using a paid cloud service is acceptable, it's worth considering. It will handle multi-columns, tables, and output structured JSON with both Arabic and English content segments. Yet, there are a few trade-offs: (1) Arabic-specific support is not guaranteed to be better in terms of character order [42] – you might still need to reorder the text in the JSON. (2) Using the API adds external dependency and cost, and might not be as straightforward to integrate into a rapid pipeline as a local library. For a one-off or highly critical project, Adobe could be the "best" in quality; for a continuous RAG pipeline dealing with many PDFs, **a self-hosted solution like PyMuPDF is typically preferred** for speed and control.

Usage Notes: Whichever tool you choose, some general best practices apply: - After extraction, **normalize the Unicode** (NFKC/NFKD) to ensure Arabic characters are in consistent form (this can fix issues like isolated vs. medial forms of letters) [39] . - Maintain metadata like page numbers and coordinates for each chunk – this helps align Arabic and English by page, and can be useful for source attribution in RAG answers. - Consider a hybrid approach: for example, use PyMuPDF to quickly parse structure and text, and if any pages fail or contain mostly tables, use a specialized table extractor or even an API on those pages. In production, having a fallback (like OCR for troublesome pages) can improve robustness [50] [51] . - Test on your actual documents. PDFs vary widely; one library might parse a particular layout better than another. For instance, if your yearbook uses very decorative Arabic fonts, an OCR-based method might ironically capture the text more correctly than text extraction (as one user found with an Arabic novel) [52] [53] . So, validate the choice on real samples.

Conclusion: For most scenarios, **PyMuPDF (MuPDF)** emerges as the best option to extract structured, bilingual text for RAG. It balances speed, accuracy, and flexibility, requiring only modest post-processing for perfect Arabic text. It has been used successfully in document QA pipelines and is well-supported. If you require higher-level chunking and can tolerate some overhead, Unstructured is worth a look (especially as it evolves). And if absolute fidelity with complex formatting is a must (and external services are on the table), Adobe's Extract API is a top-tier solution, though one that might exceed the needs of a typical RAG workflow.

Sources:

- Real-world experiences comparing PDF extractors [34] [10]
- Techniques for multi-column text detection in PyMuPDF [13]
- Arabic text extraction challenges and solutions (Stack Overflow) [4] [54]
- Unstructured library RTL text issue (GitHub) [32]
- Reddit discussion of PDF parsing for RAG (performance of pdfplumber vs PyMuPDF) [55]
- Poppler `pdftotext` success with Arabic (Stack Overflow) [39]
- Adobe Extract API language support note [42]

---

[1] How to extract text from a two-column PDF using PDFPlumber
https://stackoverflow.com/questions/68919080/how-to-extract-text-from-a-two-column-pdf-using-pdfplumber

[2] GitHub - jsvine/pdfplumber: Plumb a PDF for detailed information about each char, rectangle, line, et cetera — and easily extract text and tables.
https://github.com/jsvine/pdfplumber

[3] [10] [23] [24] [25] [26] [27] [28] [29] [30] [31] [34] [35] [46] [47] [48] [50] [51] I Tested 7 Python PDF Extractors So You Don’t Have To (2025 Edition) | by Aman Kumar | Medium
https://onlyoneaman.medium.com/i-tested-7-python-pdf-extractors-so-you-dont-have-to-2025-edition-c88013922257

[4] [5] [6] [7] [8] [9] [45] [49] [52] [53] [54] python - Extracting text from PDF in Arabic language and getting backwards text - Stack Overflow
https://stackoverflow.com/questions/75050321/extracting-text-from-pdf-in-arabic-language-and-getting-backwards-text

[11] [33] [55] Need help with PDF processing for RAG pipeline : r/Rag
https://www.reddit.com/r/Rag/comments/1itlq0d/need_help_with_pdf_processing_for_rag_pipeline/

[12] [13] [14] [15] [16] Extract Text From a Multi-Column Document | Medium
https://medium.com/@pymupdf/extract-text-from-a-multi-column-document-using-pymupdf-in-python-a0395ebc8e28

[17] [18] [19] [20] RTL (Arabic) ligatures problem when extracting text from PDF · Issue #2199 · pymupdf/PyMuPDF · GitHub
https://github.com/pymupdf/PyMuPDF/issues/2199

[21] [22] python - RTL (Arabic) ligatures problem when extracting text from PDF - Stack Overflow
https://stackoverflow.com/questions/75280067/rtl-arabic-ligatures-problem-when-extracting-text-from-pdf

[32] bug/right to left languages · Issue #3927 · Unstructured-IO/unstructured · GitHub
https://github.com/Unstructured-IO/unstructured/issues/3927

[36] [37] Arabic ligatures order issue when extracting text from PDF · Issue #850 · pdfminer/pdfminer.six · GitHub
https://github.com/pdfminer/pdfminer.six/issues/850

[38] [39] [44] Arabic pdf text extraction - Stack Overflow
https://stackoverflow.com/questions/72559699/arabic-pdf-text-extraction

[40] PDF Extract API - Adobe Developer
https://developer.adobe.com/document-services/docs/overview/pdf-extract-api/

[41] Adobe PDF Services - Workfront Fusion - Experience League
https://experienceleague.adobe.com/en/docs/workfront-fusion/using/references/apps-and-their-modules/adobe-connectors/pdf-modules

[42] PDF Extract API Language Support - Adobe Product Community
https://community.adobe.com/t5/acrobat-services-api-discussions/pdf-extract-api-language-support/td-p/12491903

[43] How do I extract Arabic Text using Acrobat DC?
https://community.adobe.com/t5/acrobat-discussions/how-do-i-extract-arabic-text-using-acrobat-dc/td-p/8781183