Docling Technical Report

Nikolaos Livathinos *, Christoph Auer *, Maksym Lysak, Ahmed Nassar, Michele Dolfi, Panagiotis Vagenas, Cesar Berrospi, Matteo Omenetti, Kasper Dinkla, Yusik Kim, Shubham Gupta, Rafael Teixeira de Lima, Valery Weber, Lucas Morin, Ingmar Meijer, Viktor Kuropiatnyk, Peter W. J. Staar

IBM Research, Rüschlikon, Switzerland
Please send correspondence to: deepsearch-core@zurich.ibm.com

Abstract

We introduce *Docling*, an easy-to-use, self-contained, MIT-licensed, open-source toolkit for document conversion, that can parse several types of popular document formats into a unified, richly structured representation. It is powered by state-of-the-art specialized AI models for layout analysis (DocLayNet) and table structure recognition (TableFormer), and runs efficiently on commodity hardware in a small resource budget. Docling is released as a Python package and can be used as a Python API or as a CLI tool. Docling's modular architecture and efficient document representation make it easy to implement extensions, new features, models, and customizations. Docling has been already integrated in other popular open-source frameworks (e.g., LlamaIndex, LangChain, spaCy), making it a natural fit for the processing of documents and the development of high-end applications. The open-source community has fully engaged in using, promoting, and developing for Docling, which gathered 10k stars on GitHub in less than a month and was reported as the No. 1 trending repository in GitHub worldwide in November 2024.

Repository — https://github.com/DS4SD/docling

1 Introduction

Converting documents back into a unified machine-processable format has been a major challenge for decades due to their huge variability in formats, weak standardization and printing-optimized characteristic, which often discards structural features and metadata. With the advent of LLMs and popular application patterns such as retrieval-augmented generation (RAG), leveraging the rich content embedded in PDFs, Office documents, and scanned document images has become ever more relevant. In the past decade, several powerful document understanding solutions have emerged on the market, most of which are commercial software, SaaS offerings on hyperscalers (Auer et al. 2022) and most recently, multimodal vision-language models. Typically, they incur a cost (e.g., for licensing or LLM inference) and cannot be run easily on local hardware. Meanwhile, only a handful of different open-source tools cover PDF, MS Word, MS PowerPoint, Images, or HTML conversion, leaving a significant feature and quality gap to proprietary solutions.

With *Docling*, we recently open-sourced a very capable and efficient document conversion tool which builds on the powerful, specialized AI models and datasets for layout analysis and table structure recognition that we developed and presented in the recent past (Livathinos et al. 2021; Pfitzmann et al. 2022; Lysak et al. 2023). Docling is designed as a simple, self-contained Python library with permissive MIT license, running entirely locally on commodity hardware. Its code architecture allows for easy extensibility and addition of new features and models. Since its launch in July 2024, Docling has attracted considerable attention in the AI developer community and ranks top on GitHub's monthly trending repositories with more than 10,000 stars at the time of writing. On October 16, 2024, Docling reached a major milestone with version 2, introducing several new features and concepts, which we outline in this updated technical report, along with details on its architecture, conversion speed benchmarks, and comparisons to other open-source assets.

The following list summarizes the features currently available on Docling:

- Parses common document formats (PDF, Images, MS Office formats, HTML) and exports to Markdown, JSON, and HTML.
- Applies advanced AI for document understanding, including detailed page layout, OCR, reading order, figure extraction, and table structure recognition.
- Establishes a unified `DoclingDocument` data model for rich document representation and operations.
- Integrates seamlessly with LlamaIndex and LangChain for generative AI applications, such as RAG.
- Offers a simple command-line interface.
- Can leverage accelerators such as GPUs.

2 State of the Art

Document conversion is a well-established field with numerous solutions already available on the market. These solutions can be categorized along several key dimensions, including open vs. closed source, permissive vs. restrictive licensing, Web APIs vs. local code deployment, susceptibility to hallucinations, conversion quality, time-to-solution, and compute resource requirements.

*These authors contributed equally.

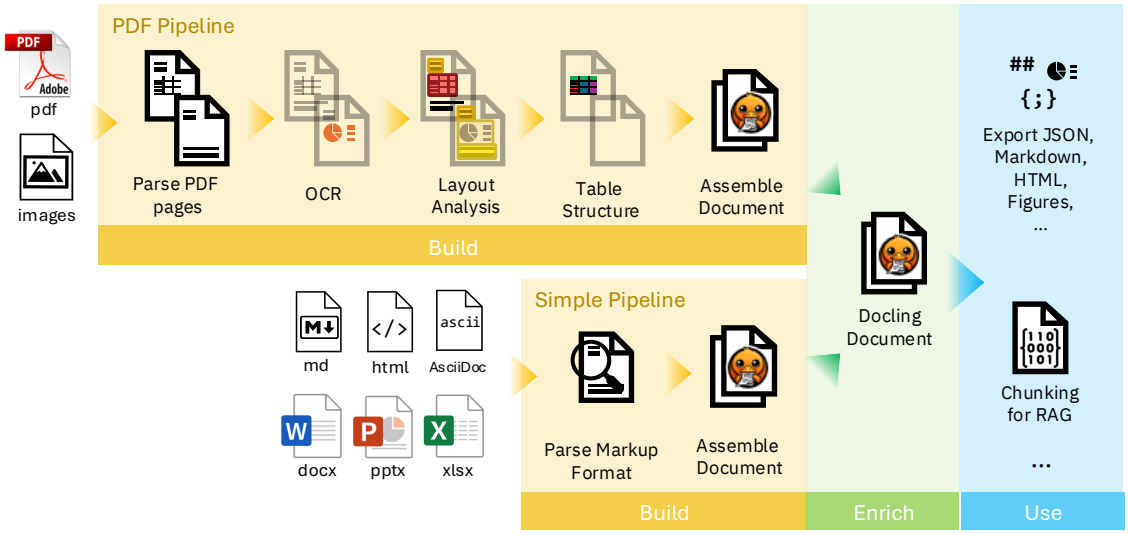


Figure 1: Sketch of Docling's pipelines and usage model. Both PDF pipeline and simple pipeline build up a `DoclingDocument` representation, which can be further enriched. Downstream applications can utilize Docling's API to inspect, export, or chunk the document for various purposes.

The most popular conversion tools today leverage visual language models (VLMs), which process page images for direct conversion. Among closed-source solutions, prominent examples include GPT-4 (OpenAI), Claude (Anthropic), and Gemini (Google). In the open-source domain, LLaVA-based models, such as LLaVA-next, are noteworthy. However, all generative AI-based models face two significant challenges. First, they are prone to hallucinations, i.e., their output may contain inaccuracies — a critical issue when faithful transcription of document content is required. Second, these models demand substantial computational resources, making the conversion process expensive. Consequently, VLM-based tools are typically offered as SaaS, with compute-intensive operations performed in the cloud.

A second class of solutions prioritizes on-premises deployment, either as Web APIs or as libraries. Examples include Adobe Acrobat, Grobid, Marker, MinerU, Unstructured, and others. These solutions often rely on multiple specialized models, such as OCR, layout analysis, and table recognition models. Docling adopts a similar approach, leveraging modular, task-specific models. This design ensures conversion without hallucinations. However, it necessitates maintaining a diverse set of models for different document components, such as formulas or figures.

Within this category, Docling distinguishes itself through its permissive MIT license, allowing organizations to integrate Docling into their solutions without incurring licensing fees or adopting restrictive licenses (e.g., GPL). Additionally, Docling offers highly accurate, resource-efficient, and fast models, making it well-suited for integration with many standard frameworks.

In summary, Docling stands out as a fundamentally hallucination-free, cost-effective, accurate, open-source library with a permissive license, offering a reliable and flexible solution for document conversion.

3 Design and Architecture

Docling is designed in a modular fashion with extensibility in mind, and it builds on three main concepts: pipelines, parser backends, and the `DoclingDocument` data model as its centerpiece (see Figure 1). Pipelines and parser backends share the responsibility of constructing and enriching a `DoclingDocument` representation from any supported input format. The `DoclingDocument` data model with its APIs enable inspection, export, and downstream processing for various applications, such as RAG.

3.1 Docling Document

Docling v2 introduces a unified document representation, `DoclingDocument`, as a Pydantic data model that can express various common document features, such as:

- Text, Tables, Pictures, Captions, Lists, and more.
- Document hierarchy with sections and groups.
- Disambiguation between main body and headers, footers (furniture).
- Layout information (i.e., bounding boxes) for all items, if available.
- Provenance information (i.e., page numbers, document origin).

With this data model, Docling enables representing document content in a unified manner, i.e., regardless of the source document format.

Besides specifying the data model, the `DoclingDocument` class defines APIs encompassing document construction, inspection, and export. Using the respective methods, users can incrementally build a `DoclingDocument`, traverse its contents in reading order, or export to commonly used formats. Docling supports lossless serialization to (and deserialization from)

JSON, and lossy export formats such as Markdown and HTML, which, unlike JSON, cannot retain all available meta information.

A `DoclingDocument` can additionally be passed to a chunker class, an abstraction that returns a stream of chunks, each of which captures some part of the document as a string accompanied by respective metadata. To enable both flexibility for downstream applications and out-of-the-box utility, Docling defines a chunker class hierarchy, providing a base type as well as specific subclasses. By using the base chunker type, downstream applications can leverage popular frameworks like LlamaIndex or LangChain, which provide a high degree of flexibility in the chunking approach. Users can therefore plug in any built-in, self-defined, or third-party chunker implementation.

3.2 Parser Backends

Document formats can be broadly categorized into two types:

1. **Low-level formats**, like PDF files or scanned images. These formats primarily encode the visual representation of the document, containing instructions for rendering text cells and lines or defining image pixels. Most semantics of the represented content are typically lost and need to be recovered through specialized AI methods, such as OCR, layout analysis, or table structure recognition.
2. **Markup-based formats**, including MS Office, HTML, Markdown, and others. These formats preserve the semantics of the content (e.g., sections, lists, tables, and figures) and are comparatively inexpensive to parse.

Docling implements several parser backends to read and interpret different formats and it routes their output to a fitting processing pipeline. For PDFs Docling provides backends which: a) retrieve all text content and their geometric properties, b) render the visual representation of each page as it would appear in a PDF viewer. For markup-based formats, the respective backends carry the responsibility of creating a `DoclingDocument` representation directly. For some formats, such as PowerPoint slides, element locations and page provenance are available, whereas in other formats (for example, MS Word or HTML), this information is unknown unless rendered in a Word viewer or a browser. The *DoclingDocument* data model handles both cases.

PDF Backends While several open-source PDF parsing Python libraries are available, in practice we ran into various limitations, among which are restrictive licensing (e.g., pymupdf (pym 2024)), poor speed, or unrecoverable quality issues, such as merged text cells across far-apart text tokens or table columns (pypdfium, PyPDF) (PyPDFium Team 2024; pypdf Maintainers 2024).

We therefore developed a custom-built PDF parser, which is based on the low-level library *qpdf* (Berkenbilt 2024). Our PDF parser is made available in a separate package named *docling-parse* and acts as the default PDF backend in Docling. As an alternative, we provide a PDF backend relying on *pypdfium* (PyPDFium Team 2024).

Other Backends Markup-based formats like HTML, Markdown, or Microsoft Office (Word, PowerPoint, Excel) as well as plain formats like AsciiDoc can be transformed directly to a `DoclingDocument` representation with the help of several third-party format parsing libraries. For HTML documents we utilize *BeautifulSoup* (Richardson 2004–2024), for Markdown we use the *Marko* library (Ming 2019–2024), and for Office XML-based formats (Word, PowerPoint, Excel) we implement custom extensions on top of the *python-docx* (Canny and contributors 2013–2024a), *python-pptx* (Canny and contributors 2013–2024b), and *openpyxl* (Eric Gazoni 2010–2024) libraries, respectively. During parsing, we identify and extract common document elements (e.g., title, headings, paragraphs, tables, lists, figures, and code) and reflect the correct hierarchy level if possible.

3.3 Pipelines

Pipelines in Docling serve as an orchestration layer which iterates through documents, gathers the extracted data from a parser backend, and applies a chain of models to: a) build up the `DoclingDocument` representation and b) enrich this representation further (e.g., classify images).

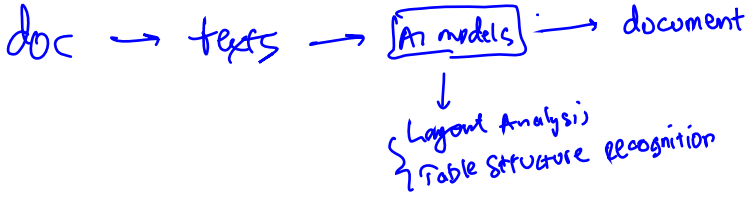
Docling provides two standard pipelines. The *StandardPdfPipeline* leverages several state-of-the-art AI models to reconstruct a high-quality *DoclingDocument* representation from PDF or image input, as described in section 4. The *SimplePipeline* handles all markup-based formats (Office, HTML, AsciiDoc) and may apply further enrichment models as well.

Pipelines can be fully customized by sub-classing from an abstract base class or cloning the default model pipeline. This effectively allows to fully customize the chain of models, add or replace models, and introduce additional pipeline configuration parameters. To create and use a custom model pipeline, you can provide a custom pipeline class as an argument to the main document conversion API.

4 PDF Conversion Pipeline

The capability to recover detailed structure and content from PDF and image files is one of Docling's defining features. In this section, we outline the underlying methods and models that drive the system.

Each document is first parsed by a PDF backend, which retrieves the programmatic text tokens, consisting of string content and its coordinates on the page, and also renders a bitmap image of each page to support downstream operations. Any image format input is wrapped in a PDF container on the fly, and proceeds through the pipeline as a scanned PDF document. Then, the standard PDF pipeline applies a sequence of AI models independently on every page of the document to extract features and content, such as layout and table structures. Finally, the results from all pages are aggregated and passed through a post-processing stage, which eventually assembles the `DoclingDocument` representation.



4.1 AI Models

As part of Docling, we release two highly capable AI models to the open-source community, which have been developed and published recently by our team. The first model is a layout analysis model, an accurate object detector for page elements (Pfitzmann et al. 2022). The second model is TableFormer (Nassar et al. 2022; Lysak et al. 2023), a state-of-the-art table structure recognition model. We provide the pre-trained weights (hosted on Hugging Face) and a separate Python package for the inference code (*docling-ibm-models*).

Layout Analysis Model Our layout analysis model is an object detector which predicts the bounding-boxes and classes of various elements on the image of a given page. Its architecture is derived from RT-DETR (Zhao et al. 2023) and re-trained on DocLayNet (Pfitzmann et al. 2022), our popular human-annotated dataset for document-layout analysis, among other proprietary datasets. For inference, our implementation relies on the *Hugging Face transformers* (Wolf et al. 2020) library and the *Safetensors* file format. All predicted bounding-box proposals for document elements are post-processed to remove overlapping proposals based on confidence and size, and then intersected with the text tokens in the PDF to group them into meaningful and complete units such as paragraphs, section titles, list items, captions, figures, or tables.

Table Structure Recognition The TableFormer model (Nassar et al. 2022), first published in 2022 and since refined with a custom structure token language (Lysak et al. 2023), is a vision-transformer model for table structure recovery. It can predict the logical row and column structure of a given table based on an input image, and determine which table cells belong to column headers, row headers or the table body. Compared to earlier approaches, TableFormer handles many characteristics of tables like partial or no borderlines, empty cells, rows or columns, cell spans and hierarchy on both column-heading and row-heading level, tables with inconsistent indentation or alignment and other complexities. For inference, our implementation relies on *PyTorch* (Ansel et al. 2024). The PDF pipeline feeds all table objects detected in the layout analysis to the TableFormer model, by providing an image-crop of the table and the included text cells. TableFormer structure predictions are matched back to the PDF cells during a post-processing step, to avoid expensive re-transcription of the table image-crop, which also makes the TableFormer model language agnostic.

OCR Docling utilizes OCR to convert scanned PDFs and extract content from bitmaps images embedded in a page. Currently, we provide integration with *EasyOCR* (eas 2024), a popular third-party OCR library with support for many languages, and Tesseract as a widely available alternative. While EasyOCR delivers reasonable transcription quality, we observe that it runs fairly slow on CPU (see section 5), making it the biggest compute expense in the pipeline.

Assembly In the final pipeline stage, Docling assembles all prediction results produced on each page into the *DoclingDocument* representation, as defined in the auxiliary Python package *docling-core*. The generated document object is passed through a post-processing model which leverages several algorithms to augment features, such as correcting the reading order or matching figures with captions.

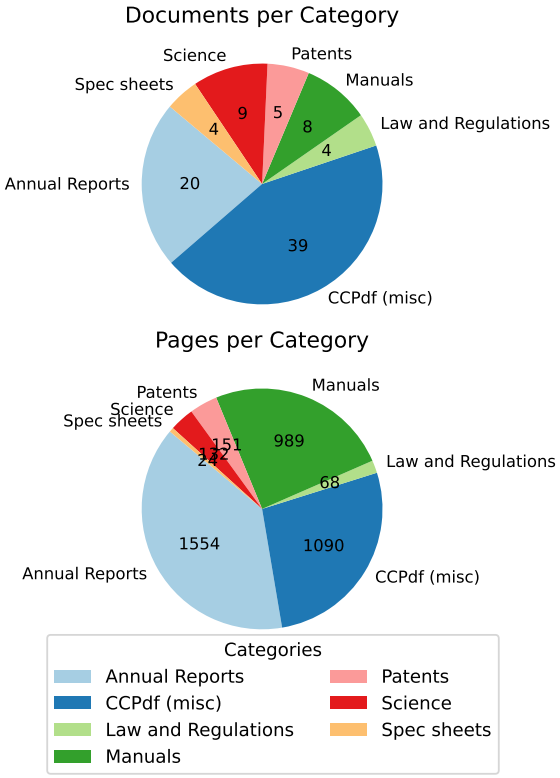


Figure 2: Dataset categories and sample counts for documents and pages.

5 Performance

In this section, we characterize the conversion speed of PDF documents with Docling in a given resource budget for different scenarios and establish reference numbers.

Further, we compare the conversion speed to three popular contenders in the open-source space, namely *unstructured.io* (Unstructured.io Team 2024), *Marker* (Paruchuri 2024), and *MinerU* (Wang et al. 2024). All aforementioned solutions can universally convert PDF documents to Markdown or similar representations and offer a library-style interface to run the document processing entirely locally. We exclude SaaS offerings and remote services for document conversion from this comparison, since the latter do not provide any possibility to control the system resources they run on, rendering any speed comparison invalid.

5.1 Benchmark Dataset

To enable a meaningful benchmark, we composed a test set of 89 PDF files covering a large variety of styles, features, content, and length (see Figure 2). This dataset is based to a large extend on our DocLayNet (Pfitzmann et al.

2022) dataset and augmented with additional samples from CCpdf (Turski et al. 2023) to increase the variety. Overall, it includes 4008 pages, 56 246 text items, 1842 tables and 4676 pictures. As such, it is large enough to provide variety without requiring excessively long benchmarking times.

5.2 System Configurations

We schedule our benchmark experiments each on two different systems to create reference numbers:

- AWS EC2 VM (g6.xlarge), 8 virtual cores (AMD EPYC 7R13, x86), 32 GB RAM, Nvidia L4 GPU (24 GB VRAM), on Ubuntu 22.04 with Nvidia CUDA 12.4 drivers
- MacBook Pro M3 Max (ARM), 64GB RAM, on macOS 14.7

All experiments on the AWS EC2 VM are carried out once with GPU acceleration enabled and once purely on the x86 CPU, resulting in three total system configurations which we refer to as M3 Max SoC, L4 GPU, and x86 CPU.

5.3 Benchmarking Methodology

We implemented several measures to enable a fair and reproducible benchmark across all tested assets. Specifically, the experimental setup accounts for the following factors:

- All assets are installed in the latest available versions, in a clean Python environment, and configured to use the state-of-the-art processing options and models, where applicable. We selectively disabled non-essential functionalities to achieve a compatible feature-set across all compared libraries.
- When running experiments on CPU, we inform all assets of the desired CPU thread budget of 8 threads, via the `OMP_NUM_THREADS` environment variable and any accepted configuration options. The L4 GPU on our AWS EC2 VM is hidden.
- When running experiments on the L4 GPU, we enable CUDA acceleration in all accepted configuration options, ensure the GPU is visible and all required runtimes for AI inference are installed with CUDA support.

Table 1 provides an overview of the versions and configuration options we considered for each asset.

5.4 Results

Runtime Characteristics To analyze Docling's runtime characteristics, we begin by exploring the relationship between document length (in pages) and conversion time. As shown in Figure 3, this relationship is not strictly linear, as documents differ in their frequency of tables and bitmap elements (i.e., scanned content). This requires OCR or table structure recognition models to engage dynamically when layout analysis has detected such elements.

By breaking down the runtimes to a page level, we receive a more intuitive measure for the conversion speed (see also Figure 4). Processing a page in our benchmark dataset requires between 0.6 sec (5^{th} percentile) and 16.3 sec (95^{th} percentile), with a median of 0.79 sec on the x86 CPU. On the

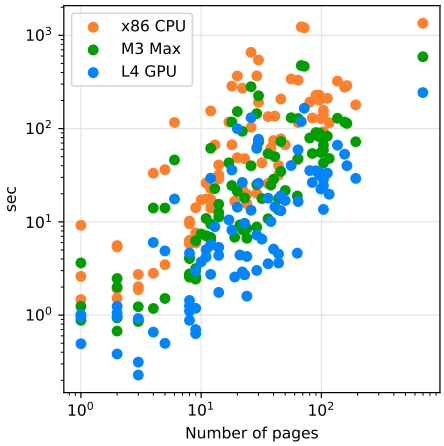


Figure 3: Distribution of conversion times for all documents, ordered by number of pages in a document, on all system configurations. Every dot represents one document. Log/log scale is used to even the spacing, since both number of pages and conversion times have long-tail distributions.

M3 Max SoC, it achieves 0.26/0.32/6.48 seconds per page (.05/median/.95), and on the Nvidia L4 GPU it achieves 57/114/2081 milliseconds per page (.05/median/.95). The large range between 5 and 95 percentiles results from the highly different complexity of content across pages (i.e., almost empty pages vs. full-page tables).

Disabling OCR saves 60% of runtime on the x86 CPU and the M3 Max SoC, and 50% on the L4 GPU. Turning off table structure recognition saves 16% of runtime on the x86 CPU and the M3 Max SoC, and 24% on the L4 GPU. Disabling both OCR and table structure recognition saves around 75% of runtime on all system configurations.

Profiling Docling's AI Pipeline We analyzed the contributions of Docling's PDF backend and all AI models in the PDF pipeline to the total conversion time. The results are shown in Figure 4. On average, processing a page took 481 ms on the L4 GPU, 3.1 s on the x86 CPU and 1.26 s on the M3 Max SoC.

It is evident that applying OCR is the most expensive operation. In our benchmark dataset, OCR engages in 578 pages. On average, transcribing a page with EasyOCR took 1.6 s on the L4 GPU, 13 s on the x86 CPU and 5 s on the M3 Max SoC. The layout model spent 44 ms on the L4 GPU, 633 ms on the x86 CPU and 271 ms on the M3 Max SoC on average for each page, making it the cheapest of the AI models, while TableFormer (fast flavour) spent 400 ms on the L4 GPU, 1.74 s on the x86 CPU and 704 ms on the M3 Max SoC on average per table. Regarding the total time spent converting our benchmark dataset, TableFormer had less impact than other AI models, since tables appeared on only 28% of all pages (see Figure 4).

On the L4 GPU, we observe a speedup of 8x (OCR), 14x (Layout model) and 4.3x (Table structure) compared to the x86 CPU and a speedup of 3x (OCR), 6x (Layout model) and 1.7x (Table structure) compared to the M3 Max CPU of

Table 1: Versions and configuration options considered for each tested asset. * denotes the default setting.

Asset	Version	OCR	Layout	Tables
Docling	2.5.2	EasyOCR*	default	TableFormer (fast)*
Marker	0.3.10	Surya*	default	default
MinerU	0.9.3	auto*	doclayout_yolo	rapid_table*
Unstructured	0.16.5	`hi_res` with table structure		

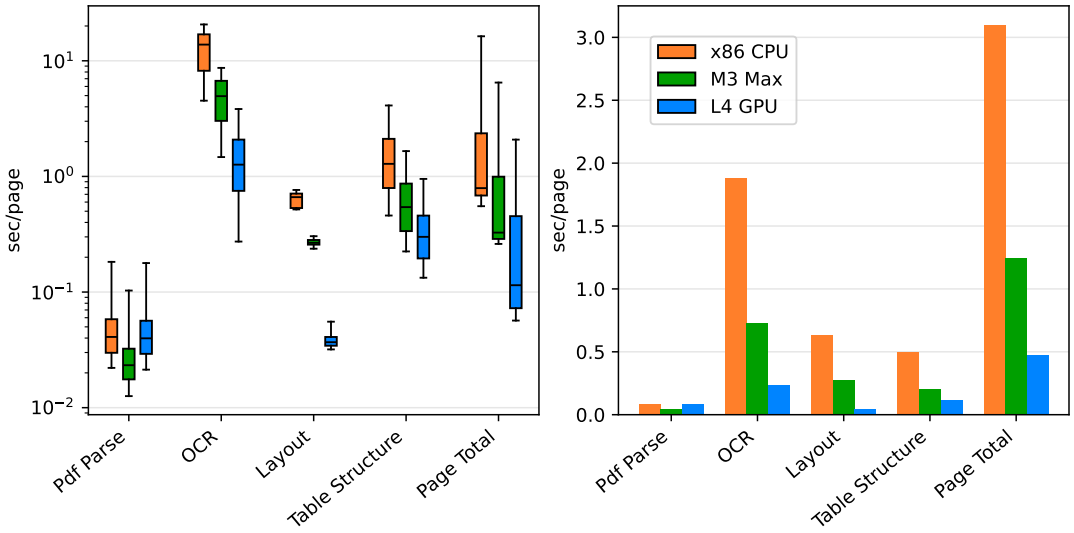


Figure 4: Contributions of PDF backend and AI models to the conversion time of a page (in seconds per page). Lower is better. Left: Ranges of time contributions for each model to pages it was applied on (i.e., OCR was applied only on pages with bitmaps, table structure was applied only on pages with tables). Right: Average time contribution to a page in the benchmark dataset (factoring in zero-time contribution for OCR and table structure models on pages without bitmaps or tables) .

our MacBook Pro. This shows that there is no equal benefit for all AI models from the GPU acceleration and there might be potential for optimization.

The time spent in parsing a PDF page through our docling-parse backend is substantially lower in comparison to the AI models. On average, parsing a PDF page took 81 ms on the x86 CPU and 44 ms on the M3 Max SoC (there is no GPU support).

Comparison to Other Tools We compare the average times to convert a page between Docling, Marker, MinerU, and Unstructured on the system configurations outlined in section 5.2. Results are shown in Figure 5.

Without GPU support, Docling leads with 3.1 sec/page (x86 CPU) and 1.27 sec/page (M3 Max SoC), followed closely by MinerU (3.3 sec/page on x86 CPU) and Unstructured (4.2 sec/page on x86 CPU, 2.7 sec/page on M3 Max SoC), while Marker needs over 16 sec/page (x86 CPU) and 4.2 sec/page (M3 Mac SoC). MinerU, despite several efforts to configure its environment, did not finish any run on our MacBook Pro M3 Max. With CUDA acceleration on the Nvidia L4 GPU, the picture changes and MinerU takes the lead over the contenders with 0.21 sec/page, compared to 0.49 sec/page with Docling and 0.86 sec/page with Marker. Unstructured does not profit from GPU acceleration.

6 Applications

Docling's document extraction capabilities make it naturally suitable for workflows like generative AI applications (e.g., RAG), data preparation for foundation model training, and fine-tuning, as well as information extraction.

As far as RAG is concerned, users can leverage existing Docling extensions for popular frameworks like LlamaIndex and then harness framework capabilities for RAG components like embedding models, vector stores, etc. These Docling extensions typically provide two modes of operation: one using a lossy export, e.g., to Markdown, and one using lossless serialization via JSON. The former provides a simple starting point, upon which any text-based chunking method may be applied (e.g., also drawing from the framework library), while the latter, which uses a swappable Docling chunker type, can be the more powerful one, as it can provide document-native RAG grounding via rich metadata such as the page number and the bounding box of the supporting context. For usage outside of these frameworks, users can still employ Docling chunkers to accelerate and simplify the development of their custom pipelines.

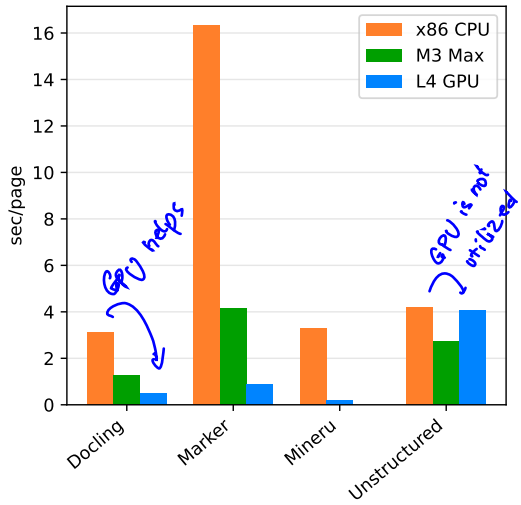


Figure 5: Conversion speed in seconds per page on our dataset in three scenarios, across all assets and system configurations. Lower bars are better. The configuration includes OCR and table structure recognition (`fast` table option on Docling and MinerU, `hi_res` in unstructured, as shown in table 1).

Besides strict RAG pipelines for Q&A, Docling can naturally be utilized in the context of broader agentic workflows for which it can provide document-based knowledge for agents to decide and act on.

Moreover, document extraction can be an invaluable component to providing ground truth data. For instance, using Docling on textbooks and research papers can significantly contribute to domain-specific knowledge when infused to foundation model training and fine-tuning.

Last but not least, Docling can be used as a backbone for information extraction tasks. Users who seek to create structured representations out of unstructured or semi-structured documents can leverage Docling for its streamlined pipeline, which maps various document formats to the standardized, unified `DoclingDocument` format, as well as its strong table understanding capabilities that can help better analyze semi-structured document parts.

7 Ecosystem

Docling is quickly evolving into a mainstream package for document conversion. The support for PDF, MS Office formats, Images, HTML, and more makes it a universal choice for downstream applications. Users appreciate the intuitiveness of the library, the high-quality, richly structured conversion output, as well as the permissive MIT license, and the possibility of running entirely locally on commodity hardware.

Among the integrations created by the Docling team and the growing community, a few are worth mentioning as depicted in Figure 6. For popular generative AI application patterns, we provide native integration within LlamaIndex (Liu 2022) and LangChain (Chase 2022) for reading documents

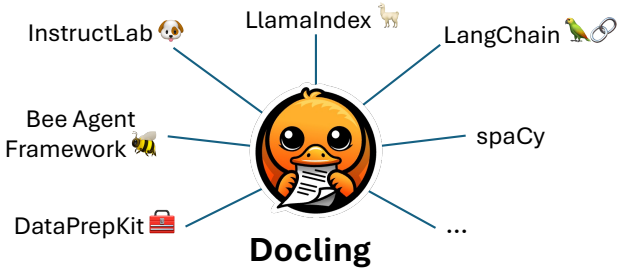


Figure 6: Ecosystem of Docling integrations contributed by the Docling team or the broader community. Docling is already used for RAG, model fine-tuning, large-scale datasets creation, information extraction and agentic workflows.

and chunking. Processing and transforming documents at scale for building large-scale multi-modal training datasets are enabled by the integration in the open IBM data-prep-kit (Wood et al. 2024). Agentic workloads can leverage the integration with the Bee framework (IBM Research 2024). For the fine-tuning of language models, Docling is integrated in InstructLab (Sudalairaj et al. 2024), where it supports the enhancement of the knowledge taxonomy.

Docling is also available and officially maintained as a system package in the Red Hat® Enterprise Linux® AI (RHEL AI) distribution, which seamlessly allows to develop, test, and run the Granite family of large language models for enterprise applications.

8 Future Work and Contributions

Docling is designed to allow an easy extension of the model library and pipelines. In the future, we plan to extend Docling with several additional models, such as a figure-classifier model, an equation-recognition model and a code-recognition model. This will help improve the quality of conversion for specific types of content, as well as augment extracted document metadata with additional information. The Docling roadmap is outlined in the discussions section [1] of the GitHub repository.

We encourage everyone to propose or implement additional features and models, and will gladly take your inputs and contributions under review. The codebase of Docling is open for use and contribution, under the MIT license agreement and in alignment with our contributing guidelines included in the Docling repository. If you use Docling in your projects, please consider citing this technical report.

References

2024. EasyOCR: Ready-to-use OCR with 80+ supported languages. https://github.com/JaidedAI/EasyOCR.

2024. PyMuPDF. https://github.com/pymupdf/PyMuPDF.

Ansel, J.; Yang, E.; He, H.; et al. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Proceed-*

[1] https://github.com/DS4SD/docling/discussions/categories/roadmap

ings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24). ACM.

Auer, C.; Dolfi, M.; Carvalho, A.; Ramis, C. B.; and Staar, P. W. 2022. Delivering Document Conversion as a Cloud Service with High Throughput and Responsiveness. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, 363–373. IEEE.

Berkenbilt, J. 2024. QPDF: A Content-Preserving PDF Document Transformer. https://github.com/qpdf/qpdf.

Canny, S.; and contributors. 2013–2024a. python-docx: Create and update Microsoft Word .docx files with Python. https://python-docx.readthedocs.io/.

Canny, S.; and contributors. 2013–2024b. python-pptx: Python library for creating and updating PowerPoint (.pptx) files. https://python-pptx.readthedocs.io/.

Chase, H. 2022. LangChain. https://github.com/langchain-ai/langchain.

Eric Gazoni, C. C. 2010–2024. openpyxl: A Python library to read/write Excel 2010 xlsx/xlsm files. https://openpyxl.readthedocs.io/.

IBM Research. 2024. Bee Agent Framework. https://github.com/i-am-bee/bee-agent-framework.

Liu, J. 2022. LlamaIndex. https://github.com/jerryjliu/llama_index.

Livathinos, N.; Berrospi, C.; Lysak, M.; Kuropiatnyk, V.; Nassar, A.; Carvalho, A.; Dolfi, M.; Auer, C.; Dinkla, K.; and Staar, P. 2021. Robust PDF Document Conversion using Recurrent Neural Networks. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(17): 15137–15145.

Lysak, M.; Nassar, A.; Livathinos, N.; Auer, C.; and Staar, P. 2023. Optimized Table Tokenization for Table Structure Recognition. In *Document Analysis and Recognition - ICDAR 2023: 17th International Conference, San José, CA, USA, August 21–26, 2023, Proceedings, Part II*, 37–50. Berlin, Heidelberg: Springer-Verlag. ISBN 978-3-031-41678-1.

Ming, F. 2019–2024. Marko: A markdown parser with high extensibility. https://github.com/frostming/marko.

Nassar, A.; Livathinos, N.; Lysak, M.; and Staar, P. 2022. Tableformer: Table structure understanding with transformers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 4614–4623.

Paruchuri, V. 2024. Marker: Convert PDF to Markdown Quickly with High Accuracy. https://github.com/VikParuchuri/marker.

Pfitzmann, B.; Auer, C.; Dolfi, M.; Nassar, A. S.; and Staar, P. 2022. DocLayNet: a large human-annotated dataset for document-layout segmentation. 3743–3751.

pypdf Maintainers. 2024. pypdf: A Pure-Python PDF Library. https://github.com/py-pdf/pypdf.

PyPDFium Team. 2024. PyPDFium2: Python bindings for PDFium. https://github.com/pypdfium2-team/pypdfium2.

Richardson, L. 2004–2024. Beautiful Soup: A Python library for parsing HTML and XML. https://www.crummy.com/software/BeautifulSoup/.

Sudalairaj, S.; Bhandwaldar, A.; Pareja, A.; Xu, K.; Cox, D. D.; and Srivastava, A. 2024. LAB: Large-Scale Alignment for ChatBots. arXiv:2403.01081.

Turski, M.; Stanisławek, T.; Kaczmarek, K.; Dyda, P.; and Graliński, F. 2023. CCpdf: Building a High Quality Corpus for Visually Rich Documents from Web Crawl Data. In Fink, G. A.; Jain, R.; Kise, K.; and Zanibbi, R., eds., *Document Analysis and Recognition - ICDAR 2023*, 348–365. Cham: Springer Nature Switzerland. ISBN 978-3-031-41682-8.

Unstructured.io Team. 2024. Unstructured.io: Open-Source Pre-Processing Tools for Unstructured Data. https://unstructured.io. Accessed: 2024-11-19.

Wang, B.; Xu, C.; Zhao, X.; Ouyang, L.; Wu, F.; Zhao, Z.; Xu, R.; Liu, K.; Qu, Y.; Shang, F.; Zhang, B.; Wei, L.; Sui, Z.; Li, W.; Shi, B.; Qiao, Y.; Lin, D.; and He, C. 2024. MinerU: An Open-Source Solution for Precise Document Content Extraction. arXiv:2409.18839.

Wolf, T.; Debut, L.; Sanh, V.; Chaumond, J.; Delangue, C.; Moi, A.; Cistac, P.; Rault, T.; Louf, R.; Funtowicz, M.; Davison, J.; Shleifer, S.; von Platen, P.; Ma, C.; Jernite, Y.; Plu, J.; Xu, C.; Scao, T. L.; Gugger, S.; Drame, M.; Lhoest, Q.; and Rush, A. M. 2020. HuggingFace's Transformers: State-of-the-art Natural Language Processing. arXiv:1910.03771.

Wood, D.; Lublinsky, B.; Roytman, A.; Singh, S.; Adam, C.; Adebayo, A.; An, S.; Chang, Y. C.; Dang, X.-H.; Desai, N.; Dolfi, M.; Emami-Gohari, H.; Eres, R.; Goto, T.; Joshi, D.; Koyfman, Y.; Nassar, M.; Patel, H.; Selvam, P.; Shah, Y.; Surendran, S.; Tsuzuku, D.; Zerfos, P.; and Daijavad, S. 2024. Data-Prep-Kit: getting your data ready for LLM application development. arXiv:2409.18164.

Zhao, Y.; Lv, W.; Xu, S.; Wei, J.; Wang, G.; Dang, Q.; Liu, Y.; and Chen, J. 2023. DETRs Beat YOLOs on Real-time Object Detection. arXiv:2304.08069.