



Python library for Agentic Document Extraction from LandingAI

🔗 landing.ai/agentic-document-extraction

📄 Apache-2.0 license

☆ 622 stars 🍴 66 forks 👁 13 watching 🔑 Branches ⚡ Activity 📋 Custom properties

📁 Tags

🌐 Public repository

🔗

🔗 5 Branches 📁 0 Tags 🔗 📁

🔍 Go to file

t



Go to file

+


Add file ▾

Code

⋮

 Landing AI Bot [skip ci] chore(release): agentic-doc 0.2.7	5905e32 · 2 days ago	
📁 .github/workflows	feat: enable integration tests (#26)	2 months ago
📁 agentic_doc	feat: change output name for extrac...	2 days ago
📁 tests	feat: change output name for extrac...	2 days ago
📄 .flake8	Initial implementation	3 months ago
📄 .gitignore	Initial implementation	3 months ago
📄 LICENSE	Initial commit	3 months ago
📄 README.md	feat: change output name for extrac...	2 days ago
📄 poetry.lock	feat: update connectors and style (#...	2 weeks ago
📄 pyproject.toml	[skip ci] chore(release): agentic-doc...	2 days ago

📖 README 📄 Apache-2.0 license

 ⋮

Agentic Document Extraction – Python Library

🔄 CI passing 💬 VisionAgent 📦 pypi package 0.2.4

[Web App](#) · [Discord](#) · [Blog](#) · [Docs](#)

Overview

The LandingAI **Agentic Document Extraction** API pulls structured data out of visually complex documents—think tables, pictures, and charts—and returns a hierarchical JSON with exact element locations.

This Python library wraps that API to provide:

- **Long-document support** – process 100+ page PDFs in a single call
- **Auto-retry / paging** – handles concurrency, time-outs, and rate limits
- **Helper utilities** – bounding-box snippets, visual debuggers, and more

Features

- 📦 **Batteries-included install:** `pip install agentic-doc` – nothing else needed → see [Installation](#)
- 📁 **All file types:** parse PDFs of *any* length, single images, or URLs → see [Supported Files](#)
- 📖 **Long-doc ready:** auto-split & parallel-process 1000+ page PDFs, then stitch results → see [Parse Large PDF Files](#)
- 🧩 **Structured output:** returns hierarchical JSON plus ready-to-render Markdown → see [Result Schema](#)
- 📷 **Ground-truth visuals:** optional bounding-box snippets and full-page visualizations → see [Save Groundings as Images](#)
- 🏃 **Batch & parallel:** feed a list; library manages threads & rate limits (`BATCH_SIZE` , `MAX_WORKERS`) → see [Parse Multiple Files in a Batch](#)
- 🔄 **Resilient:** exponential-backoff retries for 408/429/502/503/504 and rate-limit hits → see [Automatically Handle API Errors and Rate Limits with Retries](#)
- 🛠️ **Drop-in helpers:** `parse_documents` , `parse_and_save_documents` , `parse_and_save_document` → see [Main Functions](#)
- ⚙️ **Config via env / .env:** tweak parallelism, logging style, retry caps—no code changes → see [Configuration Options](#)
- 🌐 **Raw API ready:** advanced users can still hit the REST endpoint directly → see the [API Docs](#)

Quick Start

Installation

```
pip install agentic-doc
```



Requirements

- Python version 3.9, 3.10, 3.11 or 3.12
- LandingAI agentic AI API key (get the key [here](#))

Set the API Key as an Environment Variable

After you get the LandingAI agentic AI API key, set the key as an environment variable (or put it in a `.env` file):

```
export VISION_AGENT_API_KEY=<your-api-key>
```



Supported Files

The library can extract data from:

- PDFs (any length)
- Images that are supported by OpenCV-Python (i.e. the `cv2` library)
- URLs pointing to PDF or image files

Basic Usage

Extract Data from One Document

Run the following script to extract data from one document and return the results in both markdown and structured chunks.

```
from agentic_doc.parse import parse

# Parse a local file
result = parse("path/to/image.png")
print(result[0].markdown) # Get the extracted data as markdown
print(result[0].chunks) # Get the extracted data as structured chunks of content

# Parse a document from a URL
result = parse("https://example.com/document.pdf")
print(result[0].markdown)

# Legacy approach (still supported)
from agentic_doc.parse import parse_documents
results = parse_documents(["path/to/image.png"])
parsed_doc = results[0]
```



Extract Data from Multiple Documents

Run the following script to extract data from multiple documents.

```
from agentic_doc.parse import parse

# Parse multiple local files
file_paths = ["path/to/your/document1.pdf", "path/to/another/document2.pdf"]
results = parse(file_paths)
for result in results:
    print(result.markdown)

# Parse and save results to a directory
results = parse(file_paths, result_save_dir="path/to/save/results")
result_paths = []
for result in results:
    result_paths.append(result.result_path)
# result_paths: ["path/to/save/results/document1_20250313_070305.json", ...]
```



Using field extraction

```
from pydantic import BaseModel, Field
from agentic_doc.parse import parse
```



```

class ExtractedFields(BaseModel):
    employee_name: str = Field(description="the full name of the employee")
    employee_ssn: str = Field(description="the social security number of the employee")
    gross_pay: float = Field(description="the gross pay of the employee")
    employee_address: str = Field(description="the address of the employee")

results = parse("mydoc.pdf", extraction_model=ExtractedFields)
fields = results[0].extraction
print(fields.employee_name)

```

Extract Data Using Connectors

The library now supports various connectors to easily access documents from different sources:

Google Drive Connector

Prerequisites: Follow the [Google Drive API Python Quickstart](#) tutorial first to set up your credentials.

The Google Drive API quickstart will guide you through:

1. Creating a Google Cloud project
2. Enabling the Google Drive API
3. Setting up OAuth 2.0 credentials

After completing the quickstart tutorial, you can use the Google Drive connector as follows:

```

from agentic_doc.parse import parse
from agentic_doc.connectors import GoogleDriveConnectorConfig

# Using OAuth credentials file (from quickstart tutorial)
config = GoogleDriveConnectorConfig(
    client_secret_file="path/to/credentials.json",
    folder_id="your-google-drive-folder-id" # Optional
)

# Parse all documents in the folder
results = parse(config)

# Parse with filtering
results = parse(config, connector_pattern="*.pdf")

```



Amazon S3 Connector

```

from agentic_doc.parse import parse
from agentic_doc.connectors import S3ConnectorConfig

config = S3ConnectorConfig(
    bucket_name="your-bucket-name",
    aws_access_key_id="your-access-key", # Optional if using IAM roles
    aws_secret_access_key="your-secret-key", # Optional if using IAM roles
    region_name="us-east-1"
)

# Parse all documents in the bucket
results = parse(config)

```



```
# Parse documents in a specific prefix/folder
results = parse(config, connector_path="documents/")
```

Local Directory Connector

```
from agentic_doc.parse import parse
from agentic_doc.connectors import LocalConnectorConfig

config = LocalConnectorConfig()

# Parse all supported documents in a directory
results = parse(config, connector_path="/path/to/documents")

# Parse with pattern filtering
results = parse(config, connector_path="/path/to/documents", connector_pattern="*.pdf")

# Parse all supported documents in a directory recursively (search subdirectories as well)
config = LocalConnectorConfig(recursive=True)
results = parse(config, connector_path="/path/to/documents")
```



URL Connector

```
from agentic_doc.parse import parse
from agentic_doc.connectors import URLConnectorConfig

config = URLConnectorConfig(
    headers={"Authorization": "Bearer your-token"}, # Optional
    timeout=60 # Optional
)

# Parse document from URL
results = parse(config, connector_path="https://example.com/document.pdf")
```



Raw Bytes Input

```
from agentic_doc.parse import parse

# Load a PDF or image file as bytes
with open("document.pdf", "rb") as f:
    raw_bytes = f.read()

# Parse the document from bytes
results = parse(raw_bytes)
```



You can also parse image bytes:

```
with open("image.png", "rb") as f:
    image_bytes = f.read()

results = parse(image_bytes)
```



This is useful when documents are already loaded into memory (e.g., from an API response or uploaded via a web interface). The parser will auto-detect the file type from the bytes.

Why Use It?

- **Simplified Setup:** No need to manage API keys or handle low-level REST calls.
- **Automatic Large File Processing:** Splits large PDFs into manageable parts and processes them in parallel.
- **Built-In Error Handling:** Automatically retries requests with exponential backoff and jitter for common HTTP errors.
- **Parallel Processing:** Efficiently parse multiple documents at once with configurable parallelism.

Main Features

With this library, you can do things that are otherwise hard to do with the Agentic Document Extraction API alone. This section describes some of the key features this library offers.

Parse Large PDF Files

A single REST API call can only handle up to certain amount of pages at a time (see [rate limits](#)). This library automatically splits a large PDF into multiple calls, uses a thread pool to process the calls in parallel, and stitches the results back together as a single result.

We've used this library to successfully parse PDFs that are 1000+ pages long.

Parse Multiple Files in a Batch

You can parse multiple files in a single function call with this library. The library processes files in parallel.

NOTE: You can change the parallelism by setting the `batch_size` setting.

Save Groundings as Images

The library can extract and save the visual regions (groundings) of the document where each chunk of content was found. This is useful for visualizing exactly what parts of the document were extracted and for debugging extraction issues.

Each grounding represents a bounding box in the original document, and the library can save these regions as individual PNG images. The images are organized by page number and chunk ID.

Here's how to use this feature:

```
from agentic_doc.parse import parse_documents

# Save groundings when parsing a document
results = parse_documents(
    ["path/to/document.pdf"],
    grounding_save_dir="path/to/save/groundings"
)

# The grounding images will be saved to:
# path/to/save/groundings/document_TIMESTAMP/page_X/CHUNK_TYPE_CHUNK_ID_Y.png
# Where X is the page number, CHUNK_ID is the unique ID of each chunk,
# and Y is the index of the grounding within the chunk
```



```
# Each chunk's grounding in the result will have the image_path set
for chunk in results[0].chunks:
    for grounding in chunk.grounding:
        if grounding.image_path:
            print(f"Grounding saved to: {grounding.image_path}")
```

This feature works with all parsing functions: `parse_documents` , `parse_and_save_documents` , and `parse_and_save_document` .

Visualize Parsing Result

The library provides a visualization utility that creates annotated images showing where each chunk of content was extracted from the document. This is useful for:

- Verifying the accuracy of the extraction
- Debugging extraction issues

Here's how to use the visualization feature:

```
from agentic_doc.parse import parse
from agentic_doc.utils import viz_parsed_document
from agentic_doc.config import VisualizationConfig

# Parse a document
results = parse("path/to/document.pdf")
parsed_doc = results[0]

# Create visualizations with default settings
# The output images have a PIL.Image.Image type
images = viz_parsed_document(
    "path/to/document.pdf",
    parsed_doc,
    output_dir="path/to/save/visualizations"
)

# Or customize the visualization appearance
viz_config = VisualizationConfig(
    thickness=2, # Thicker bounding boxes
    text_bg_opacity=0.8, # More opaque text background
    font_scale=0.7, # Larger text
    # Custom colors for different chunk types
    color_map={
        ChunkType.TITLE: (0, 0, 255), # Red for titles
        ChunkType.TEXT: (255, 0, 0), # Blue for regular text
        # ... other chunk types ...
    }
)

images = viz_parsed_document(
    "path/to/document.pdf",
    parsed_doc,
    output_dir="path/to/save/visualizations",
    viz_config=viz_config
)

# The visualization images will be saved as:
```



```
# path/to/save/visualizations/document_viz_page_X.png
# Where X is the page number
```

The visualization shows:

- Bounding boxes around each extracted chunk
- Chunk type and index labels
- Different colors for different types of content (titles, text, tables, etc.)
- Semi-transparent text backgrounds for better readability

Automatically Handle API Errors and Rate Limits with Retries

The REST API endpoint imposes rate limits per API key. This library automatically handles the rate limit error or other intermittent HTTP errors with retries.

For more information, see [Error Handling](#) and [Configuration Options](#).

Error Handling

This library implements a retry mechanism for handling API failures:

- Retries are performed for these HTTP status codes: 408, 429, 502, 503, 504.
- Exponential backoff with jitter is used for retry wait time.
- The initial retry wait time is 1 second, which increases exponentially.
- Retry will stop after `max_retries` attempts. Exceeding the limit raises an exception and results in a failure for this request.
- Retry wait time is capped at `max_retry_wait_time` seconds.
- Retries include a random jitter of up to 10 seconds to distribute requests and prevent the thundering herd problem.

Parsing Errors

If the REST API request encounters an unrecoverable error during parsing (either from client-side or server-side), the library includes an [errors](#) field in the final result for the affected page(s). Each error contains the error message, `error_code` and corresponding page number.

Configuration Options

The library uses a [Settings](#) object to manage configuration. You can customize these settings either through environment variables or a `.env` file:

Below is an example `.env` file that customizes the configurations:

```
# Number of files to process in parallel, defaults to 4
BATCH_SIZE=4
# Number of threads used to process parts of each file in parallel, defaults to 5.
MAX_WORKERS=2
# Maximum number of retry attempts for failed intermittent requests, defaults to 100
MAX_RETRIES=80
# Maximum wait time in seconds for each retry, defaults to 60
MAX_RETRY_WAIT_TIME=30
```




```
# Logging style for retry, defaults to log_msg
RETRY_LOGGING_STYLE=log_msg
```

Max Parallelism

The maximum number of parallel requests is determined by multiplying `BATCH_SIZE` × `MAX_WORKERS` .

NOTE: The maximum parallelism allowed by this library is 100.

Specifically, increasing `MAX_WORKERS` can speed up the processing of large individual files, while increasing `BATCH_SIZE` improves throughput when processing multiple files.

NOTE: Your job's maximum processing throughput may be limited by your API rate limit. If your rate limit isn't high enough, you may encounter rate limit errors, which the library will automatically handle through retries.

The optimal values for `MAX_WORKERS` and `BATCH_SIZE` depend on your API rate limit and the latency of each REST API call. For example, if your account has a rate limit of 5 requests per minute, and each REST API call takes approximately 60 seconds to complete, and you're processing a single large file, then `MAX_WORKERS` should be set to 5 and `BATCH_SIZE` to 1.

You can find your REST API latency in the logs. If you want to increase your rate limit, schedule a time to meet with us [here](#).

Set `RETRY_LOGGING_STYLE`

The `RETRY_LOGGING_STYLE` setting controls how the library logs the retry attempts.

- `log_msg` : Log the retry attempts as a log messages. Each attempt is logged as a separate message. This is the default setting.
- `inline_block` : Print a yellow progress block ('█') on the same line. Each block represents one retry attempt. Choose this if you don't want to see the verbose retry logging message and still want to track the number of retries that have been made.
- `none` : Do not log the retry attempts.

Troubleshooting & FAQ

Common Issues

- **API Key Errors:**
Ensure your API key is correctly set as an environment variable.
- **Rate Limits:**
The library automatically retries requests if you hit the API rate limit. Adjust `BATCH_SIZE` or `MAX_WORKERS` if you encounter frequent rate limit errors.
- **Parsing Failures:**
If a document fails to parse, an error chunk will be included in the result, detailing the error message and page index.
- **URL Access Issues:** If you're having trouble accessing documents from URLs, check that the URLs are publicly accessible and point to supported file types (PDF or images).

Note on `include_marginalia` and `include_metadata_in_markdown`

- `include_marginalia` : If True, the parser will attempt to extract and include marginalia (footer notes, page number, etc.) from the document in the output.
- `include_metadata_in_markdown` : If True, the output markdown will include metadata.

Both parameters default to True. You can set them to False to exclude these elements from the output.

Example: Using the new parameters

```
from agentic_doc.parse import parse
```



Releases

No releases published

Packages

No packages published

Contributors 11



Languages

● Python 100.0%