PaperQA

PaperQA2

GITHUB ↗ pypi package 5.20.0 ↗ Lint and Test passing ↗ License Apache 2.0
python 3.10 | 3.11 | 3.12

PaperQA2 is a package for doing high-accuracy retrieval augmented generation (RAG) on PDFs or text files,
with a focus on the scientific literature.
See our recent 2024 paper ↗ to see examples of PaperQA2's superhuman performance in scientific tasks like
question answering, summarization, and contradiction detection.

Quickstart

In this example we take a folder of research paper PDFs,
magically get their metadata - including citation counts with a retraction check,
then parse and cache PDFs into a full-text search index,
and finally answer the user question with an LLM agent.

```
pip install paper-qa
cd my_papers
pqa ask 'How can carbon nanotubes be manufactured at a large scale?'
```

Example Output

Question: Has anyone designed neural networks that compute with proteins or DNA?

> The claim that neural networks have been designed to compute with DNA is supported by multiple sources. The work by Qian, Winfree, and Bruck demonstrates the use of DNA strand displacement cascades to construct neural network components, such as artificial neurons and associative memories, using a DNA-based system (Qian2011Neural pages 1-2, Qian2011Neural pages 15-16, Qian2011Neural pages 54-56). This research includes the implementation of a 3-bit XOR gate and a four-neuron Hopfield associative memory, showcasing the potential of DNA for neural network computation. Additionally, the application of deep learning techniques to genomics, which involves computing with DNA sequences, is well-documented. Studies have applied convolutional neural networks (CNNs) to predict genomic features such as transcription factor binding and DNA accessibility (Eraslan2019Deep pages 4-5, Eraslan2019Deep pages 5-6). These models leverage DNA sequences as input data, effectively using neural networks to compute with DNA. While the provided excerpts do not explicitly mention protein-based neural network computation, they do highlight the use of neural networks in tasks related to protein sequences, such as predicting DNA-protein binding (Zeng2016Convolutional pages 1-2). However, the primary focus remains on DNA-based computation.

What is PaperQA2

PaperQA2 is engineered to be the best agentic RAG model for working with scientific papers.
Here are some features:

- A simple interface to get good answers with grounded responses containing in-text citations.
- State-of-the-art implementation including document metadata-awareness in embeddings and LLM-based re-ranking and contextual summarization (RCS).

- Support for agentic RAG, where a language agent can iteratively refine queries and answers.
- Automatic redundant fetching of paper metadata, including citation and journal quality data from multiple providers.
- A usable full-text search engine for a local repository of PDF/text files.
- A robust interface for customization, with default support for all LiteLLM ↗ models.

By default, it uses OpenAI embeddings ↗ and models ↗ with a Numpy vector DB to embed and search documents. However, you can easily use other closed-source, open-source models or embeddings (see details below).

PaperQA2 depends on some awesome libraries/APIs that make our repo possible.
Here are some in no particular order:

1. Semantic Scholar ↗
2. Crossref ↗
3. Unpaywall ↗
4. Pydantic ↗
5. tantivy ↗
6. LiteLLM ↗
7. pybtex ↗
8. PyMuPDF ↗

PaperQA2 vs PaperQA

We've been working on hard on fundamental upgrades for a while and mostly followed SemVer ↗.
meaning we've incremented the major version number on each breaking change.
This brings us to the current major version number v5.
So why call is the repo now called PaperQA2?
We wanted to remark on the fact though that we've exceeded human performance on many important metrics ↗.
So we arbitrarily call version 5 and onward PaperQA2,
and versions before it as PaperQA1 to denote the significant change in performance.

We recognize that we are challenged at naming and counting at FutureHouse,
so we reserve the right at any time to arbitrarily change the name to PaperCrow.

What's New in Version 5 (aka PaperQA2)?

Version 5 added:

- A CLI `pqa`
- Agentic workflows invoking tools for
  paper search, gathering evidence, and generating an answer
- Removed much of the statefulness from the `Docs` object
- A migration to LiteLLM for compatibility with many LLM providers
  as well as centralized rate limits and cost tracking
- A bundled set of configurations (read here))
  containing known-good hyperparameters

Note that `Docs` objects pickled from prior versions of `PaperQA` are incompatible with version 5,
and will need to be rebuilt.
Also, our minimum Python version was increased to Python 3.11.

PaperQA2 Algorithm

To understand PaperQA2, let's start with the pieces of the underlying algorithm.
The default workflow of PaperQA2 is as follows:

Phase	PaperQA2 Actions
1. Paper Search	- Get candidate papers from LLM-generated keyword query
	- Chunk, embed, and add candidate papers to state
2. Gather Evidence	- Embed query into vector

	- Rank top *k* document chunks in current state
	- Create scored summary of each chunk in the context of the current query
	- Use LLM to re-score and select most relevant summaries
3. Generate Answer	- Put best summaries into prompt with context

The tools can be invoked in any order by a language agent.
For example, an LLM agent might do a narrow and broad search,
or using different phrasing for the gather evidence step from the generate answer step.

Installation

For a non-development setup,
install PaperQA2 (aka version 5) from PyPI ↗.
Note version 5 requires Python 3.11+.

```
pip install paper-qa>=5
```

For development setup,
please refer to the CONTRIBUTING.md file.

PaperQA2 uses an LLM to operate,
so you'll need to either set an appropriate API key environment variable ↗ (i.e. `export OPENAI_API_KEY=sk-...`)
or set up an open source LLM server (i.e. using llamafile ↗.
Any LiteLLM compatible model can be configured to use with PaperQA2.

If you need to index a large set of papers (100+),
you will likely want an API key for both Crossref ↗ and Semantic Scholar ↗,
which will allow you to avoid hitting public rate limits using these metadata services.
Those can be exported as `CROSSREF_API_KEY` and `SEMANTIC_SCHOLAR_API_KEY` variables.

CLI Usage

The fastest way to test PaperQA2 is via the CLI. First navigate to a directory with some papers and use the `pqa` cli:

```
$ pqa ask 'What manufacturing challenges are unique to bispecific antibodies?'
```

You will see PaperQA2 index your local PDF files, gathering the necessary metadata for each of them (using Crossref ↗ and Semantic Scholar ↗),
search over that index, then break the files into chunked evidence contexts, rank them, and ultimately generate an answer. The next time this directory is queried, your index will already be built (save for any differences detected, like new added papers), so it will skip the indexing and chunking steps.

All prior answers will be indexed and stored, you can view them by querying via the `search` subcommand, or access them yourself in your `PQA_HOME` directory, which defaults to `~/.pqa/`.

```
$ pqa search -i 'answers' 'antibodies'
```

PaperQA2 is highly configurable, when running from the command line, `pqa --help` shows all options and short descriptions. For example to run with a higher temperature:

```
$ pqa --temperature 0.5 ask 'What manufacturing challenges are unique to bispecific antibodies?'
```

You can view all settings with `pqa view`. Another useful thing is to change to other templated settings - for example `fast` is a setting that answers more quickly and you can see it with `pqa -s fast view`

Maybe you have some new settings you want to save? You can do that with

```
pqa -s my_new_settings --temperature 0.5 --llm foo-bar-5 save
```

and then you can use it with

```
pqa -s my_new_settings ask 'What manufacturing challenges are unique to
bispecific antibodies?'
```

If you run `pqa` with a command which requires a new indexing, say if you change the default chunk_size, a new index will automatically be created for you.

```
pqa --parsing.chunk_size 5000 ask 'What manufacturing challenges are
unique to bispecific antibodies?'
```

You can also use `pqa` to do full-text search with use of LLMs view the search command. For example, let's save the index from a directory and give it a name:

```
pqa -i nanomaterials index
```

Now I can search for papers about thermoelectrics:

```
pqa -i nanomaterials search thermoelectrics
```

or I can use the normal ask

```
pqa -i nanomaterials ask 'Are there nm scale features in thermoelectric
materials?'
```

Both the CLI and module have pre-configured settings based on prior performance and our publications, they can be invoked as follows:

```
pqa --settings <setting name> ask 'Are there nm scale features in
thermoelectric materials?'
```

Bundled Settings

Inside [`paperqa/configs`] ↗ we bundle known useful settings:

Setting Name	Description
high_quality	Highly performant, relatively expensive (due to having `evidence_k` = 15) query using a `ToolSelector` agent.
fast	Setting to get answers cheaply and quickly.
wikicrow	Setting to emulate the Wikipedia article writing used in our WikiCrow publication.
contracrow	Setting to find contradictions in papers, your query should be a claim that needs to be flagged as a contradiction (or not).
debug	Setting useful solely for debugging, but not in any actual application beyond debugging.
tier1_limits	Settings that match OpenAI rate limits for each tier, you can use `tier<1-5>_limits` to specify the tier.

Rate Limits

If you are hitting rate limits, say with the OpenAI Tier 1 plan, you can add them into PaperQA2.
For each OpenAI tier, a pre-built setting exists to limit usage.

```
pqa --settings 'tier1_limits' ask 'Are there nm scale features in
thermoelectric materials?'
```

This will limit your system to use the tier1_limits ↗,
and slow down your queries to accommodate.

You can also specify them manually with any rate limit string that matches the specification in the limits ↗ module:

```
pqa --summary_llm_config '{"rate_limit": {"gpt-4o-2024-11-20": "30000
per 1 minute"}}' ask 'Are there nm scale features in thermoelectric
materials?'
```

Or by adding into a `Settings` object, if calling imperatively:

```
from paperqa import Settings, ask

answer_response = ask(
    "What manufacturing challenges are unique to bispecific
antibodies?",
    settings=Settings(
        llm_config={"rate_limit": {"gpt-4o-2024-11-20": "30000 per 1
minute"}},
        summary_llm_config={"rate_limit": {"gpt-4o-2024-11-20": "30000
per 1 minute"}},
    ),
)
```

Library Usage

PaperQA2's full workflow can be accessed via Python directly:

```
from paperqa import Settings, ask

answer_response = ask(
    "What manufacturing challenges are unique to bispecific
antibodies?",
    settings=Settings(temperature=0.5, paper_directory="my_papers"),
)
```

Please see our installation docs for how to install the package from PyPI.

Agentic Adding/Querying Documents

The answer object has the following attributes: `formatted_answer`, `answer` (answer alone), `question` , and `context` (the summaries of passages found for answer). `ask` will use the `SearchPapers` tool, which will query a local index of files, you can specify this location via the `Settings` object:

```
from paperqa import Settings, ask

answer_response = ask(
    "What manufacturing challenges are unique to bispecific
antibodies?",
    settings=Settings(temperature=0.5, paper_directory="my_papers"),
)
```

`ask` is just a convenience wrapper around the real entrypoint, which can be accessed if you'd like to run concurrent asynchronous workloads:

```
from paperqa import Settings, agent_query

answer_response = await agent_query(
    query="What manufacturing challenges are unique to bispecific
antibodies?",
    settings=Settings(temperature=0.5, paper_directory="my_papers"),
)
```

The default agent will use an LLM based agent,
but you can also specify a `"fake"` agent to use a hard coded call path of search -> gather evidence -> answer to reduce token usage.

Manual (No Agent) Adding/Querying Documents

Normally via agent execution, the agent invokes the search tool,
which adds documents to the `Docs` object for you behind the scenes.
However, if you prefer fine-grained control,
you can directly interact with the `Docs` object.

Note that manually adding and querying `Docs` does not impact performance.
It just removes the automation associated with an agent picking the documents to add.

```python
from paperqa import Docs, Settings

# valid extensions include .pdf, .txt, .md, and .html
doc_paths = ("myfile.pdf", "myotherfile.pdf")

# Prepare the Docs object by adding a bunch of documents
docs = Docs()
for doc_path in doc_paths:
    await docs.aadd(doc_path)

# Set up how we want to query the Docs object
settings = Settings()
settings.llm = "claude-3-5-sonnet-20240620"
settings.answer.answer_max_sources = 3

# Query the Docs object to get an answer
session = await docs.aquery(
    "What manufacturing challenges are unique to bispecific
antibodies?",
    settings=settings,
)
print(session)
```

Async

PaperQA2 is written to be used asynchronously.
The synchronous API is just a wrapper around the async.
Here are the methods and their `async` equivalents:

Sync	Async
`Docs.add`	`Docs.aadd`
`Docs.add_file`	`Docs.aadd_file`
`Docs.add_url`	`Docs.aadd_url`
`Docs.get_evidence`	`Docs.aget_evidence`
`Docs.query`	`Docs.aquery`

The synchronous version just calls the async version in a loop.
Most modern python environments support `async` natively (including Jupyter notebooks!).
So you can do this in a Jupyter Notebook:

```
import asyncio
from paperqa import Docs


async def main() -> None:
    docs = Docs()
    # valid extensions include .pdf, .txt, .md, and .html
    for doc in ("myfile.pdf", "myotherfile.pdf"):
        await docs.aadd(doc)

    session = await docs.aquery(
        "What manufacturing challenges are unique to bispecific
antibodies?"
    )
    print(session)


asyncio.run(main())
```

Choosing Model

By default, PaperQA2 uses OpenAI's `gpt-4o-2024-11-20` model for the `summary_llm`, `llm`, and `agent_llm`.
Please see the Settings Cheatsheet
for more information on these settings.
PaperQA2 also defaults to using OpenAI's `text-embedding-3-small` model for the `embedding` setting.
If you don't have an OpenAI API key, you can use a different embedding model.
More information about embedding models can be found in the "Embedding Model" section.

We use the `lmi` ↗ package for our LLM interface,
which in turn uses `litellm` to support many LLM providers.
You can adjust this easily to use any model supported by `litellm`:

```python
from paperqa import Settings, ask

answer_response = ask(
    "What manufacturing challenges are unique to bispecific
antibodies?",
    settings=Settings(
        llm="gpt-4o-mini", summary_llm="gpt-4o-mini",
paper_directory="my_papers"
    ),
)
```

To use Claude, make sure you set the `ANTHROPIC_API_KEY` environment variable.
In this example, we also use a different embedding model.
Please make sure to `pip install paper-qa[local]` to use a local embedding model.

```python
from paperqa import Settings, ask
from paperqa.settings import AgentSettings

answer_response = ask(
    "What manufacturing challenges are unique to bispecific
antibodies?",
    settings=Settings(
        llm="claude-3-5-sonnet-20240620",
        summary_llm="claude-3-5-sonnet-20240620",
        agent=AgentSettings(agent_llm="claude-3-5-sonnet-20240620"),
        # SEE: https://huggingface.co/sentence-transformers/multi-qa-
MiniLM-L6-cos-v1
        embedding="st-multi-qa-MiniLM-L6-cos-v1",
    ),
)
```

Or Gemini, by setting the `GEMINI_API_KEY` from Google AI Studio

```
from paperqa import Settings, ask
from paperqa.settings import AgentSettings

answer_response = ask(
    "What manufacturing challenges are unique to bispecific
antibodies?",
    settings=Settings(
        llm="gemini/gemini-2.0-flash",
        summary_llm="gemini/gemini-2.0-flash",
        agent=AgentSettings(agent_llm="gemini/gemini-2.0-flash"),
        embedding="gemini/text-embedding-004",
    ),
)
```

Locally Hosted

You can use llama.cpp to be the LLM. Note that you should be using relatively large models, because PaperQA2 requires following a lot of instructions. You won't get good performance with 7B models.

The easiest way to get set-up is to download a llama file ↗ and execute it with `-cb -np 4 -a my-llm-model --embedding` which will enable continuous batching and embeddings.

```python
from paperqa import Settings, ask

local_llm_config = dict(
    model_list=[
        dict(
            model_name="my_llm_model",
            litellm_params=dict(
                model="my-llm-model",
                api_base="http://localhost:8080/v1",
                api_key="sk-no-key-required",
                temperature=0.1,
                frequency_penalty=1.5,
                max_tokens=512,
            ),
        )
    ]
)

answer_response = ask(
    "What manufacturing challenges are unique to bispecific
antibodies?",
    settings=Settings(
        llm="my-llm-model",
        llm_config=local_llm_config,
        summary_llm="my-llm-model",
        summary_llm_config=local_llm_config,
    ),
)
```

Models hosted with `ollama` are also supported.

To run the example below make sure you have downloaded llama3.2 and mxbai-embed-large via ollama.

```
from paperqa import Settings, ask

local_llm_config = {
    "model_list": [
        {
            "model_name": "ollama/llama3.2",
            "litellm_params": {
                "model": "ollama/llama3.2",
                "api_base": "http://localhost:11434",
            },
        }
    ]
}

answer_response = ask(
    "What manufacturing challenges are unique to bispecific
antibodies?",
    settings=Settings(
        llm="ollama/llama3.2",
        llm_config=local_llm_config,
        summary_llm="ollama/llama3.2",
        summary_llm_config=local_llm_config,
        embedding="ollama/mxbai-embed-large",
    ),
)
```

Embedding Model

Embeddings are used to retrieve k texts (where k is specified via `Settings.answer.evidence_k`) for re-ranking and contextual summarization.
If you don't want to use embeddings, but instead just fetch all chunks, disable "evidence retrieval" via the `Settings.answer.evidence_retrieval` setting.

PaperQA2 defaults to using OpenAI (`text-embedding-3-small`) embeddings, but has flexible options for both vector stores and embedding choices.

Specifying the Embedding Model

The simplest way to specify the embedding model is via `Settings.embedding`:

```
from paperqa import Settings, ask

answer_response = ask(
    "What manufacturing challenges are unique to bispecific
antibodies?",
    settings=Settings(embedding="text-embedding-3-large"),
)
```

`embedding` accepts any embedding model name supported by litellm. PaperQA2 also supports an embedding input of `"hybrid-<model_name>"` i.e. `"hybrid-text-embedding-3-small"` to use a hybrid sparse keyword (based on a token modulo embedding) and dense vector embedding, where any litellm model can be used in the dense model name. `"sparse"` can be used to use a sparse keyword embedding only.

Embedding models are used to create PaperQA2's index of the full-text embedding vectors (`texts_index` argument). The embedding model can be specified as a setting when you are adding new papers to the `Docs` object:

```
from paperqa import Docs, Settings

docs = Docs()
for doc in ("myfile.pdf", "myotherfile.pdf"):
    await docs.aadd(doc, settings=Settings(embedding="text-embedding-
large-3"))
```

Note that PaperQA2 uses Numpy as a dense vector store.
Its design of using a keyword search initially reduces the number of chunks needed for each answer to a relatively small number < 1k.
Therefore, `NumpyVectorStore` is a good place to start, it's a simple in-memory store, without an index.
However, if a larger-than-memory vector store is needed, you can an external vector database like Qdrant ↗ via the `QdrantVectorStore` class.

The hybrid embeddings can be customized:

```
from paperqa import (
    Docs,
    HybridEmbeddingModel,
    SparseEmbeddingModel,
    LiteLLMEmbeddingModel,
)


model = HybridEmbeddingModel(
    models=[LiteLLMEmbeddingModel(), SparseEmbeddingModel(ndim=1024)]
)
docs = Docs()
for doc in ("myfile.pdf", "myotherfile.pdf"):
    await docs.aadd(doc, embedding_model=model)
```

The sparse embedding (keyword) models default to having 256 dimensions, but this can be specified via the `ndim` argument.

Local Embedding Models (Sentence Transformers)

You can use a `SentenceTransformerEmbeddingModel` model if you install `sentence-transformers`, which is a local embedding library ↗ with support for HuggingFace models and more. You can install it by adding the `local` extras.

```
pip install paper-qa[local]
```

and then prefix embedding model names with `st-`:

```
from paperqa import Settings, ask

answer_response = ask(
    "What manufacturing challenges are unique to bispecific
antibodies?",
    settings=Settings(embedding="st-multi-qa-MiniLM-L6-cos-v1"),
)
```

or with a hybrid model

```
from paperqa import Settings, ask

answer_response = ask(
    "What manufacturing challenges are unique to bispecific
antibodies?",
    settings=Settings(embedding="hybrid-st-multi-qa-MiniLM-L6-cos-v1"),
)
```

Adjusting number of sources

You can adjust the numbers of sources (passages of text) to reduce token usage or add more context. `k` refers to the top k most relevant and diverse (may from different sources) passages. Each passage is sent to the LLM to summarize, or determine if it is irrelevant. After this step, a limit of `max_sources` is applied so that the final answer can fit into the LLM context window. Thus, `k` > `max_sources` and `max_sources` is the number of sources used in the final answer.

```
from paperqa import Settings

settings = Settings()
settings.answer.answer_max_sources = 3
settings.answer.k = 5

await docs.aquery(
    "What manufacturing challenges are unique to bispecific
antibodies?",
    settings=settings,
)
```

Using Code or HTML

You do not need to use papers -- you can use code or raw HTML. Note that this tool is focused on answering questions, so it won't do well at writing code. One note is that the tool cannot infer citations from code, so you will need to provide them yourself.

```
import glob
import os
from paperqa import Docs

source_files = glob.glob("**/*.js")

docs = Docs()
for f in source_files:
    # this assumes the file names are unique in code
    await docs.aadd(f, citation="File " + os.path.basename(f),
docname=os.path.basename(f))
session = await docs.aquery("Where is the search bar in the header
defined?")
print(session)
```

Using External DB/Vector DB and Caching

You may want to cache parsed texts and embeddings in an external database or file. You can then build a Docs object from those directly:

```
from paperqa import Docs, Doc, Text

docs = Docs()

for ... in my_docs:
    doc = Doc(docname=..., citation=..., dockey=..., citation=...)
    texts = [Text(text=..., name=..., doc=doc) for ... in my_texts]
    docs.add_texts(texts, doc)
```

Creating Index

Indexes will be placed in the home directory ↗ by default.
This can be controlled via the `PQA_HOME` environment variable.

Indexes are made by reading files in the `Settings.paper_directory`.
By default, we recursively read from subdirectories of the paper directory, unless disabled using `Settings.index_recursively`.
The paper directory is not modified in any way, it's just read from.

Manifest Files

The indexing process attempts to infer paper metadata like title and DOI
using LLM-powered text processing.
You can avoid this point of uncertainty using a "manifest" file,
which is a CSV containing three columns (order doesn't matter):

- `file_location`: relative path to the paper's PDF within the index directory
- `doi`: DOI of the paper
- `title`: title of the paper

By providing this information,
we ensure queries to metadata providers like Crossref are accurate.

Reusing Index

The local search indexes are built based on a hash of the current `Settings` object.
So make sure you properly specify the `paper_directory` to your `Settings` object.
In general, it's advisable to:

1. Pre-build an index given a folder of papers (can take several minutes)
2. Reuse the index to perform many queries

```
import os

from paperqa import Settings
from paperqa.agents.main import agent_query
from paperqa.agents.search import get_directory_index


async def amain(folder_of_papers: str | os.PathLike) -> None:
    settings = Settings(paper_directory=folder_of_papers)

    # 1. Build the index. Note an index name is autogenerated when
unspecified
    built_index = await get_directory_index(settings=settings)
    print(settings.get_index_name())  # Display the autogenerated index
name
    print(await built_index.index_files)  # Display the index contents

    # 2. Use the settings as many times as you want with ask
    answer_response_1 = await agent_query(
        query="What is the best way to make a vaccine?",
        settings=settings,
    )
    answer_response_2 = await agent_query(
        query="What manufacturing challenges are unique to bispecific
antibodies?",
        settings=settings,
    )
```

Using Clients Directly

One of the most powerful features of PaperQA2 is its ability to combine data from multiple metadata sources. For example, Unpaywall ↗ can provide open access status/direct links to PDFs, Crossref ↗ can provide bibtex, and Semantic Scholar ↗ can provide citation licenses. Here's a short demo of how to do this:

```
from paperqa.clients import DocMetadataClient, ALL_CLIENTS

client = DocMetadataClient(clients=ALL_CLIENTS)
details = await client.query(title="Augmenting language models with
chemistry tools")

print(details.formatted_citation)
# Andres M. Bran, Sam Cox, Oliver Schilter, Carlo Baldassari, Andrew D.
White, and Philippe Schwaller.
#  Augmenting large language models with chemistry tools. Nature
Machine Intelligence,
# 6:525-535, May 2024. URL: https://doi.org/10.1038/s42256-024-00832-8,
# doi:10.1038/s42256-024-00832-8.
# This article has 243 citations and is from a domain leading peer-
reviewed journal.

print(details.citation_count)
# 243

print(details.license)
# cc-by

print(details.pdf_url)
# https://www.nature.com/articles/s42256-024-00832-8.pdf
```

the `client.query` is meant to check for exact matches of title. It's a bit robust (like to casing, missing a word). There are duplicates for titles though - so you can also add authors to disambiguate. Or you can provide a doi directly `client.query(doi="10.1038/s42256-024-00832-8")`.

If you're doing this at a large scale, you may not want to use `ALL_CLIENTS` (just omit the argument) and you can specify which specific fields you want to speed up queries. For example:

```
details = await client.query(
    title="Augmenting large language models with chemistry tools",
    authors=["Andres M. Bran", "Sam Cox"],
    fields=["title", "doi"],
)
```

will return much faster than the first query and we'll be certain the authors match.

Settings Cheatsheet

Setting	Default	Description
`llm`	`"gpt-4o-2024-11-20"`	Default LLM for most things, including answers. Should be 'best' LLM.
`llm_config`	`None`	Optional configuration for `llm`.
`summary_llm`	`"gpt-4o-2024-11-20"`	Default LLM for summaries and parsing citations.
`summary_llm_config`	`None`	Optional configuration for `summary_llm`.
`embedding`	`"text-embedding-3-small"`	Default embedding model for texts.
`embedding_config`	`None`	Optional configuration for `embedding`.
`temperature`	`0.0`	Temperature for LLMs.
`batch_size`	`1`	Batch size for calling LLMs.
`texts_index_mmr_lambda`	`1.0`	Lambda for MMR in text index.
`verbosity`	`0`	Integer verbosity level for logging (0-3). 3 = all LLM/Embeddings calls logged.
`answer.evidence_k`	`10`	Number of evidence pieces to retrieve.
`answer.evidence_detailed_citations`	`True`	Include detailed citations in summaries.
`answer.evidence_retrieval`	`True`	Use retrieval vs processing all docs.
`answer.evidence_summary_length`	`"about 100 words"`	Length of evidence summary.

`answer.evidence_skip_summary`	`False`	Whether to skip summarization.
`answer.answer_max_sources`	`5`	Max number of sources for an answer.
`answer.max_answer_attempts`	`None`	Max attempts to generate an answer.
`answer.answer_length`	`"about 200 words, but can be longer"`	Length of final answer.
`answer.max_concurrent_requests`	`4`	Max concurrent requests to LLMs.
`answer.answer_filter_extra_background`	`False`	Whether to cite background info from model.
`answer.get_evidence_if_no_contexts`	`True`	Allow lazy evidence gathering.
`parsing.chunk_size`	`5000`	Characters per chunk (0 for no chunking).
`parsing.page_size_limit`	`1,280,000`	Character limit per page.
`parsing.use_doc_details`	`True`	Whether to get metadata details for docs.
`parsing.overlap`	`250`	Characters to overlap chunks.
`parsing.defer_embedding`	`False`	Whether to defer embedding until summarization.
`parsing.chunking_algorithm`	`ChunkingOptions.SIMPLE_OVERLAP`	Algorithm for chunking.
`parsing.doc_filters`	`None`	Optional filters for allowed documents.
`parsing.use_human_readable_clinical_trials`	`False`	Parse clinical trial JSONs into readable text.
`prompt.summary`	`summary_prompt`	Template for summarizing text, must contain variables matching `summary_prompt`.

`prompt.qa`	`qa_prompt`	Template for QA, must contain variables matching `qa_prompt`.
`prompt.select`	`select_paper_prompt`	Template for selecting papers, must contain variables matching `select_paper_prompt`.
`prompt.pre`	`None`	Optional pre-prompt templated with just the original question to append information before a qa prompt.
`prompt.post`	`None`	Optional post-processing prompt that can access PQASession fields.
`prompt.system`	`default_system_prompt`	System prompt for the model.
`prompt.use_json`	`True`	Whether to use JSON formatting.
`prompt.summary_json`	`summary_json_prompt`	JSON-specific summary prompt.
`prompt.summary_json_system`	`summary_json_system_prompt`	System prompt for JSON summaries.
`prompt.context_outer`	`CONTEXT_OUTER_PROMPT`	Prompt for how to format all contexts in generate answer.
`prompt.context_inner`	`CONTEXT_INNER_PROMPT`	Prompt for how to format a single context in generate answer. Must contain 'name' and 'text' variables.
`agent.agent_llm`	`"gpt-4o-2024-11-20"`	Model to use for agent making tool selections.
`agent.agent_llm_config`	`None`	Optional configuration for `agent_llm`.
`agent.agent_type`	`"ToolSelector"`	Type of agent to use.

`agent.agent_config`	`None`	Optional kwarg for AGENT constructor.
`agent.agent_system_prompt`	`env_system_prompt`	Optional system prompt message.
`agent.agent_prompt`	`env_reset_prompt`	Agent prompt.
`agent.return_paper_metadata`	`False`	Whether to include paper title/year in search tool results.
`agent.search_count`	`8`	Search count.
`agent.timeout`	`500.0`	Timeout on agent execution (seconds).
`agent.should_pre_search`	`False`	Whether to run search tool before invoking agent.
`agent.tool_names`	`None`	Optional override on tools to provide the agent.
`agent.max_timesteps`	`None`	Optional upper limit on environment steps.
`agent.index.name`	`None`	Optional name of the index.
`agent.index.paper_directory`	`Current working directory`	Directory containing papers to be indexed.
`agent.index.manifest_file`	`None`	Path to manifest CSV with document attributes.
`agent.index.index_directory`	`pqa_directory("indexes")`	Directory to store PQA indexes.
`agent.index.use_absolute_paper_directory`	`False`	Whether to use absolute paper directory path.
`agent.index.recurse_subdirectories`	`True`	Whether to recurse into subdirectories when indexing.

Where do I get papers?

Well that's a really good question! It's probably best to just download PDFs of papers you think will help answer your question and start from there.

See detailed docs about zotero, openreview and parsing

Callbacks

To execute a function on each chunk of LLM completions, you need to provide a function that can be executed on each chunk. For example, to get a typewriter view of the completions, you can do:

```
from paperqa import Docs


def typewriter(chunk: str) -> None:
    print(chunk, end="")


docs = Docs()

# add some docs...

await docs.aquery(
    "What manufacturing challenges are unique to bispecific
antibodies?",
    callbacks=[typewriter],
)
```

Caching Embeddings

In general, embeddings are cached when you pickle a `Docs` regardless of what vector store you use. So as long as you save your underlying `Docs` object, you should be able to avoid re-embedding your documents.

Customizing Prompts

You can customize any of the prompts using settings.

```
from paperqa import Docs, Settings

my_qa_prompt = (
    "Answer the question '{question}'\n"
    "Use the context below if helpful. "
    "You can cite the context using the key like (Example2012). "
    "If there is insufficient context, write a poem "
    "about how you cannot answer.\n\n"
    "Context: {context}"
)

docs = Docs()
settings = Settings()
settings.prompts.qa = my_qa_prompt
await docs.aquery("Are covid-19 vaccines effective?",
settings=settings)
```

Pre and Post Prompts

Following the syntax above, you can also include prompts that
are executed after the query and before the query. For example, you can use this to critique the answer.

FAQ

How come I get different results than your papers?

Internally at FutureHouse, we have a slightly different set of tools. We're trying to get some of them, like citation traversal, into this repo. However, we have APIs and licenses to access research papers that we cannot share openly. Similarly, in our research papers' results we do not start with the known relevant PDFs. Our agent has to identify them using keyword search over all papers, rather than just a subset. We're gradually aligning these two versions of PaperQA, but until there is an open-source way to freely access papers (even just open source papers) you will need to provide PDFs yourself.

How is this different from LlamaIndex or LangChain?

LangChain ↗
and LlamaIndex ↗
are both frameworks for working with LLM applications,
with abstractions made for agentic workflows and retrieval augmented generation.

Over time, the PaperQA team over time chose to become framework-agnostic,
instead outsourcing LLM drivers to LiteLLM ↗
and no framework besides Pydantic for its tools.
PaperQA focuses on scientific papers and their metadata.

PaperQA can be reimplemented using either LlamaIndex or LangChain.
For example, our `GatherEvidence` tool can be reimplemented
as a retriever with an LLM-based re-ranking and contextual summary.
There is similar work with the tree response method in LlamaIndex.

Can I save or load?

The `Docs` class can be pickled and unpickled. This is useful if you want to save the embeddings of the documents and then load them later.

```
import pickle

# save
with open("my_docs.pkl", "wb") as f:
    pickle.dump(docs, f)

# load
with open("my_docs.pkl", "rb") as f:
    docs = pickle.load(f)
```

Reproduction

Contained in docs/2024-10-16_litqa2-splits.json5 ↗
are the question IDs used in train, evaluation, and test splits,
as well as paper DOIs used to build the splits' indexes.

- Train and eval splits: question IDs come fromLAB-Bench's LitQA2 question IDs ↗.
- Test split: questions IDs come fromaviary-paper-data's LitQA2 question IDs ↗.

There are multiple papers slowly building PaperQA, shown below in Citation.
To reproduce:

- `skarlinski2024language` : train and eval splits are applicable.
  The test split remains held out.
- `narayanan2024aviarytraininglanguageagents` : train, eval, and test splits are applicable.

Example on how to use LitQA for evaluation can be found inaviary.litqa ↗.

Citation

Please read and cite the following papers if you use this software:

```
@article{narayanan2024aviarytraininglanguageagents,
      title = {Aviary: training language agents on challenging
scientific tasks},
      author = {
      Siddharth Narayanan and
 James D. Braza and
 Ryan-Rhys Griffiths and
 Manu Ponnapati and
 Albert Bou and
 Jon Laurent and
 Ori Kabeli and
 Geemi Wellawatte and
 Sam Cox and
 Samuel G. Rodriques and
 Andrew D. White},
      journal = {arXiv preprent arXiv:2412.21154},
      year = {2024},
      url = {https://doi.org/10.48550/arXiv.2412.21154},
}
```

```
@article{skarlinski2024language,
    title = {Language agents achieve superhuman synthesis of scientific
knowledge},
    author = {
    Michael D. Skarlinski and
 Sam Cox and
 Jon M. Laurent and
 James D. Braza and
 Michaela Hinks and
 Michael J. Hammerling and
 Manvitha Ponnapati and
 Samuel G. Rodriques and
 Andrew D. White},
    journal = {arXiv preprent arXiv:2409.13740},
    year = {2024},
    url = {https://doi.org/10.48550/arXiv.2409.13740}
}
```

```
@article{lala2023paperqa,
    title = {PaperQA: Retrieval-Augmented Generative Agent for
Scientific Research},
    author = {
    Jakub Lála and
 Odhran O'Donoghue and
 Aleksandar Shtedritski and
 Sam Cox and
 Samuel G. Rodriques and
 Andrew D. White},
    journal = {arXiv preprint arXiv:2312.07559},
    year = {2023},
    url = {https://doi.org/10.48550/arXiv.2312.07559}
}
```

Contributing to PaperQA

Thank you for your interest in contributing to PaperQA! Here are some guidelines to help you get started.

Setting up the development environment

We use `uv` for our local development.

1. Install `uv` by following the instructions on the uv website.
2. Run the following command to install all dependencies and set up the development environment:

```
uv sync
```

Installing the package for development

If you prefer to use `pip` for installing the package in development mode, you can do so by running:

```
pip install -e ".[dev]"
```

Where the `dev` extra includes development dependencies such as `pytest`.

Running tests and other tooling

Use the following commands:

- Run tests (requires an OpenAI key in your environment)

```
pytest
# or for multiprocessing based parallelism
pytest -n auto
```

- Run `pre-commit` for formatting and type checking

  ```
  pre-commit run --all-files
  ```

- Run `mypy`, `refurb`, or `pylint` directly:

  ```
  mypy paperqa
  # or
  refurb paperqa
  # or
  pylint paperqa
  ```

See our GitHub Actions `tests.yml` ↗ for further reference.

Using `pytest-recording` and VCR cassettes

We use the `pytest-recording` ↗ plugin to create VCR cassettes to cache HTTP requests, making our unit tests more deterministic.

To record a new VCR cassette:

```
uv run pytest --record-mode=once tests/desired_test_module.py
```

And the new cassette(s) should appear in `tests/cassettes` ↗.

Our configuration for `pytest-recording` can be found in `tests/conftest.py` ↗. This includes header removals (e.g. OpenAI `authorization` key) from responses to ensure sensitive information is excluded from the cassettes.

Please ensure cassettes are less than 1 MB to keep tests loading quickly.

Happy coding!

docs

tutorials

PaperQA2 for Clinical Trials

PaperQA2 now natively supports querying clinical trials in addition to any documents supplied by the user. It uses a new tool, the aptly named `clinical_trials_search` tool. Users don't have to provide any clinical trials to the tool itself, it uses the `clinicaltrials.gov` API to retrieve them on the fly. As of January 2025, the tool is not enabled by default, but it's easy to configure. Here's an example where we query only clinical trials, without using any documents:

```
from paperqa import Settings, agent_query

answer_response = await agent_query(
    query="What drugs have been found to effectively treat Ulcerative
Colitis?",
    settings=Settings.from_name("search_only_clinical_trials"),
)

print(answer_response.session.answer)
```

Output

```
Several drugs have been found to effectively treat Ulcerative Colitis
(UC),
targeting different mechanisms of the disease.

Golimumab, a tumor necrosis factor (TNF) inhibitor marketed as
Simponi®, has demonstrated efficacy
in treating moderate-to-severe UC. Administered subcutaneously, it was
shown to maintain clinical
response through Week 54 in patients, as assessed by the Partial Mayo
Score (NCT02092285).

Mesalazine, an anti-inflammatory drug, is commonly used for UC
treatment. In a study comparing
mesalazine enemas to faecal microbiota transplantation (FMT) for left-
sided UC,
mesalazine enemas (4g daily) were effective in inducing clinical
remission (Mayo score ≤ 2) (NCT03104036).

Antibiotics have also shown potential in UC management. A combination
of doxycycline,
amoxicillin, and metronidazole induced remission in 60-70% of patients
with moderate-to-severe
UC in prior studies. These antibiotics are thought to alter gut
microbiota, reducing pathobionts
 and promoting beneficial bacteria (NCT02217722, NCT03986996).

Roflumilast, a phosphodiesterase-4 (PDE4) inhibitor, is being
investigated for mild-to-moderate UC.
Preliminary findings suggest it may improve disease severity and
biochemical markers when
added to conventional treatments (NCT05684484).

These treatments highlight diverse therapeutic approaches, including
immunosuppression,
microbiota modulation, and anti-inflammatory mechanisms.
```

You can see the in-line citations for each clinical trial used as a response for each query. If you'd like to see more data on the specific contexts that were used to answer the query:

```
print(answer_response.session.contexts)
```

```
[Context(context='The excerpt mentions that a search on
ClinicalTrials.gov for clinical trials related to drugs
treating Ulcerative Colitis yielded 689 trials. However, it does not
provide specific information about which
drugs have been found effective for treating Ulcerative Colitis.',
text=Text(text='', name=...
```

Using `Settings.from_name('search_only_clinical_trials')` is a shortcut, but note that you can easily add `clinical_trial_search` into any custom `Settings` by just explicitly naming it as a tool:

```
from pathlib import Path
from paperqa import Settings, agent_query, AgentSetting
from paperqa.agents.tools import DEFAULT_TOOL_NAMES

# you can start with the default list of PaperQA tools
print(DEFAULT_TOOL_NAMES)
# >>> ['paper_search', 'gather_evidence', 'gen_answer', 'reset',
'complete'],

# we can start with a directory with a potentially useful paper in it
print(list(Path("my_papers").iterdir()))

# now let's query using standard tools + clinical_trials
answer_response = await agent_query(
    query="What drugs have been found to effectively treat Ulcerative
Colitis?",
    settings=Settings(
        paper_directory="my_papers",
        agent={"tool_names": DEFAULT_TOOL_NAMES +
["clinical_trials_search"]},
    ),
)

# let's check out the formatted answer (with references included)
print(answer_response.session.formatted_answer)
```

Question: What drugs have been found to effectively treat Ulcerative Colitis?

Several drugs have been found effective in treating Ulcerative Colitis (UC), with treatment
strategies varying based on disease severity and extent. For mild-to-moderate UC, 5-aminosalicylic
acid (5-ASA) is the first-line therapy. Topical 5-ASA, such as mesalazine suppositories (1 g/day),
is effective for proctitis or distal colitis, inducing remission in 31-80% of patients. Oral mesalazine
at higher doses (e.g., 4.8 g/day) can accelerate clinical improvement in more extensive disease
(meier2011currenttreatmentof pages 1-2; meier2011currenttreatmentof pages 3-4).

For moderate-to-severe cases, corticosteroids are commonly used. Oral steroids like prednisolone
(40-60 mg/day) or intravenous steroids such as methylprednisolone (60 mg/day) and hydrocortisone
(400 mg/day) are standard for inducing remission (meier2011currenttreatmentof pages 3-4). Tumor
necrosis factor (TNF)-α blockers, such as infliximab, are effective for steroid-refractory cases
(meier2011currenttreatmentof pages 2-3; meier2011currenttreatmentof pages 3-4).

Immunosuppressive agents, including azathioprine and 6-mercaptopurine, are used for maintenance
therapy in steroid-dependent or refractory cases (meier2011currenttreatmentof pages 2-3;
meier2011currenttreatmentof pages 3-4). Antibiotics, such as combinations of penicillin,
tetracycline, and metronidazole, have shown promise in altering the microbiota and inducing
remission in some patients, though their efficacy varies (NCT02217722).

References

1. (meier2011currenttreatmentof pages 2-3): Johannes Meier and Andreas Sturm. Current treatment
of ulcerative colitis. World journal of gastroenterology, 17 27:3204-12, 2011.
URL: https://doi.org/10.3748/wjg.v17.i27.3204,
doi:10.3748/wjg.v17.i27.3204.

2. (meier2011currenttreatmentof pages 3-4): Johannes Meier and Andreas Sturm. Current treatment

```
of ulcerative colitis. World journal of gastroenterology, 17 27:3204-
12, 2011. URL:
https://doi.org/10.3748/wjg.v17.i27.3204, doi:10.3748/wjg.v17.i27.3204.

3. (NCT02217722): Prof. Arie Levine. Use of the Ulcerative Colitis Diet
for Induction of
Remission. Prof. Arie Levine. 2014. ClinicalTrials.gov Identifier:
NCT02217722

4. (meier2011currenttreatmentof pages 1-2): Johannes Meier and Andreas
Sturm. Current
treatment of ulcerative colitis. World journal of gastroenterology, 17
27:3204-12, 2011.
 URL: https://doi.org/10.3748/wjg.v17.i27.3204,
doi:10.3748/wjg.v17.i27.3204.
```

We now see both papers and clinical trials cited in our response. For convenience, we have a `Settings.from_name` that works as well:

```
from paperqa import Settings, agent_query

answer_response = await agent_query(
    query="What drugs have been found to effectively treat Ulcerative
Colitis?",
    settings=Settings.from_name("clinical_trials"),
)
```

And, this works with the `pqa` cli as well:

```
>>> pqa --settings 'search_only_clinical_trials' ask 'what is Ibuprofen
effective at treating?'
```

```
...
[13:29:50] Completing 'what is Ibuprofen effective at treating?' as
'certain'.
        Answer: Ibuprofen is a non-steroidal anti-inflammatory drug
(NSAID) effective
        in treating various conditions, including pain, inflammation,
and fever.
        It is widely used for tension-type
        headaches, with studies showing that ibuprofen sodium provides
significant
        pain relief and reduces pain intensity compared to standard
ibuprofen and placebo
        over a 3-hour period (NCT01362491).
        Intravenous ibuprofen is effective in managing postoperative
pain, particularly
        in orthopedic surgeries, and helps control the inflammatory
process. When combined
        with opioids, it reduces opioid
        consumption and associated side effects, making it a key
component of
        multimodal analgesia (NCT05401916, NCT01773005).

        Ibuprofen is also effective in pediatric populations as a
first-line
        anti-inflammatory and antipyretic agent due to its relatively
        low adverse effects compared to other NSAIDs (NCT01478022).
        Additionally, it has been studied for its potential use in
managing
        chronic periodontitis through subgingival irrigation with a 2%
ibuprofen
        mouthwash, which reduces periodontal pocket depth and
        bleeding on probing, improving periodontal health
(NCT02538237).

        These findings highlight ibuprofen's versatility in treating
pain, inflammation,
        fever, and specific conditions like tension headaches,
postoperative pain, and periodontal diseases.
```

Measuring PaperQA2 with LFRQA

> This tutorial is available as a Jupyter notebook here

Overview

The **LFRQA dataset** was introduced in the paper *RAG-QA Arena: Evaluating Domain Robustness for Long-Form Retrieval-Augmented Question Answering* ↗. It features **1,404 science questions** (along with other categories) that have been human-annotated with answers. This tutorial walks through the process of setting up the dataset for use and benchmarking.

Download the Annotations

First, we need to obtain the annotated dataset from the official repository:

```
# Create a new directory for the dataset
!mkdir -p data/rag-qa-benchmarking

# Get the annotated questions
!curl https://raw.githubusercontent.com/awslabs/rag-qa-
arena/refs/heads/main/data/\
annotations_science_with_citation.jsonl \
-o data/rag-qa-benchmarking/annotations_science_with_citation.jsonl
```

Download the Robust-QA Documents

LFRQA is built upon **Robust-QA**, so we must download the relevant documents:

```
# Download the Lotte dataset, which includes the required documents
!curl
https://downloads.cs.stanford.edu/nlp/data/colbert/colbertv2/lotte.tar.
gz --output lotte.tar.gz

# Extract the dataset
!tar -xvzf lotte.tar.gz

# Move the science test collection to our dataset folder
!cp lotte/science/test/collection.tsv ./data/rag-qa-
benchmarking/science_test_collection.tsv

# Clean up unnecessary files
!rm lotte.tar.gz
!rm -rf lotte
```

For more details, refer to the original paper: *RAG-QA Arena: Evaluating Domain Robustness for Long-Form Retrieval-Augmented Question Answering ↗*.

Load the Data

We now load the documents into a pandas dataframe:

```
import os

import pandas as pd

# Load questions and answers dataset
rag_qa_benchmarking_dir = os.path.join("data", "rag-qa-benchmarking")

# Load documents dataset
lfrqa_docs_df = pd.read_csv(
    os.path.join(rag_qa_benchmarking_dir,
"science_test_collection.tsv"),
    sep="\t",
    names=["doc_id", "doc_text"],
)
```

Select the Documents to Use

RobustQA consists on 1.7M documents. Hence, it takes around 3 hours to build the whole index.

To run a test, we can use 1% of the dataset. This will be accomplished by selecting the first 1% available documents and the questions referent to these documents.

```
proportion_to_use = 1 / 100
amount_of_docs_to_use = int(len(lfrqa_docs_df) * proportion_to_use)
print(f"Using {amount_of_docs_to_use} out of {len(lfrqa_docs_df)}
documents")
```

Prepare the Document Files

We now create the document directory and store each document as a separate text file, so that paperqa can build the index.

```
partial_docs = lfrqa_docs_df.head(amount_of_docs_to_use)
lfrqa_directory = os.path.join(rag_qa_benchmarking_dir, "lfrqa")
os.makedirs(
    os.path.join(lfrqa_directory, "science_docs_for_paperqa", "files"),
exist_ok=True
)

for i, row in partial_docs.iterrows():
    doc_id = row["doc_id"]
    doc_text = row["doc_text"]

    with open(
        os.path.join(
            lfrqa_directory, "science_docs_for_paperqa", "files", f"
{doc_id}.txt"
        ),
        "w",
        encoding="utf-8",
    ) as f:
        f.write(doc_text)

    if i % int(len(partial_docs) * 0.05) == 0:
        progress = (i + 1) / len(partial_docs)
        print(f"Progress: {progress:.2%}")
```

Create the Manifest File

The **manifest file** keeps track of document metadata for the dataset. We need to fill some fields so that paperqa doesn't try to get metadata using llm calls. This will make the indexing process faster.

```
manifest = partial_docs.copy()
manifest["file_location"] = manifest["doc_id"].apply(lambda x:
f"files/{x}.txt")
manifest["doi"] = ""
manifest["title"] = manifest["doc_id"]
manifest["key"] = manifest["doc_id"]
manifest["docname"] = manifest["doc_id"]
manifest["citation"] = "_"
manifest = manifest.drop(columns=["doc_id", "doc_text"])
manifest.to_csv(
    os.path.join(lfrqa_directory, "science_docs_for_paperqa",
"manifest.csv"),
    index=False,
)
```

Filter and Save Questions

Finally, we load the questions and filter them to ensure we only include questions that reference the selected documents:

```
questions_df = pd.read_json(
    os.path.join(rag_qa_benchmarking_dir,
"annotations_science_with_citation.jsonl"),
    lines=True,
)
partial_questions = questions_df[
    questions_df.gold_doc_ids.apply(
        lambda ids: all(_id < amount_of_docs_to_use for _id in ids)
    )
]
partial_questions.to_csv(
    os.path.join(lfrqa_directory, "questions.csv"),
    index=False,
)

print("Using", len(partial_questions), "questions")
```

Install paperqa

From now on, we will be using the paperqa library, so we need to install it:

```
!pip install paper-qa
```

Index the Documents

Now we will build an index for the LFRQA documents. The index is a **Tantivy index**, which is a fast, full-text search engine library written in Rust. Tantivy is designed to handle large datasets efficiently, making it ideal for searching through a vast collection of papers or documents.

Feel free to adjust the concurrency settings as you like. Because we defined a manifest, we don’t need any API keys for building this index because we don't discern any citation metadata, but you do need LLM API keys to answer questions.

Remember that this process is quick for small portions of the dataset, but can take around 3 hours for the whole dataset.

```
import nest_asyncio

nest_asyncio.apply()
```

We add the line above to handle async code within a notebook.

However, to improve compatibility and speed up the indexing process, we strongly recommend running the following code in a separate `.py` file

```
import os

from paperqa import Settings
from paperqa.agents import build_index
from paperqa.settings import AgentSettings, IndexSettings,
ParsingSettings

settings = Settings(
    agent=AgentSettings(
        index=IndexSettings(
            name="lfrqa_science_index",
            paper_directory=os.path.join(
                "data", "rag-qa-benchmarking", "lfrqa",
"science_docs_for_paperqa"
            ),
            index_directory=os.path.join(
                "data", "rag-qa-benchmarking", "lfrqa",
"science_docs_for_paperqa_index"
            ),
            manifest_file="manifest.csv",
            concurrency=10_000,
            batch_size=10_000,
        )
    ),
    parsing=ParsingSettings(
        use_doc_details=False,
        defer_embedding=True,
    ),
)

build_index(settings=settings)
```

After this runs, you will have an index ready to use!

Benchmark!

After you have built the index, you are ready to run the benchmark. We advice running this in a separate `.py` file.

To run this, you will need to have the `ldp` ↗ and `fhaviary[lfrqa]` ↗ packages installed.

```
!pip install ldp "fhaviary[lfrqa]"
```

```
import asyncio
import json
import logging
import os

import pandas as pd
from aviary.envs.lfrqa import LFRQAQuestion, LFRQATaskDataset
from ldp.agent import SimpleAgent
from ldp.alg.runners import Evaluator, EvaluatorConfig

from paperqa import Settings
from paperqa.settings import AgentSettings, IndexSettings

logging.basicConfig(level=logging.ERROR)

log_results_dir = os.path.join("data", "rag-qa-benchmarking",
"results")
os.makedirs(log_results_dir, exist_ok=True)


async def log_evaluation_to_json(
    lfrqa_question_evaluation: dict,
) -> None:  # noqa: RUF029
    json_path = os.path.join(
        log_results_dir, f"{lfrqa_question_evaluation['qid']}.json"
    )
    with open(json_path, "w") as f:  # noqa: ASYNC230
        json.dump(lfrqa_question_evaluation, f, indent=2)


async def evaluate() -> None:
    settings = Settings(
        agent=AgentSettings(
            index=IndexSettings(
                name="lfrqa_science_index",
                paper_directory=os.path.join(
                    "data", "rag-qa-benchmarking", "lfrqa",
"science_docs_for_paperqa"
                ),
                index_directory=os.path.join(
                    "data",
                    "rag-qa-benchmarking",
                    "lfrqa",
                    "science_docs_for_paperqa_index",
                ),
            )
        )
    )
```

```
    ,

    data: list[LFRQAQuestion] = [
        LFRQAQuestion(**row)
        for row in pd.read_csv(
            os.path.join("data", "rag-qa-benchmarking", "lfrqa",
"questions.csv")
        )[["qid", "question", "answer",
"gold_doc_ids"]].to_dict(orient="records")
    ]

    dataset = LFRQATaskDataset(
        data=data,
        settings=settings,
        evaluation_callback=log_evaluation_to_json,
    )

    evaluator = Evaluator(
        config=EvaluatorConfig(batch_size=3),
        agent=SimpleAgent(),
        dataset=dataset,
    )
    await evaluator.evaluate()


if __name__ == "__main__":
    asyncio.run(evaluate())
```

After running this, you can find the results in the `data/rag-qa-benchmarking/results` folder. Here is an example of how to read them:

```
import glob

json_files = glob.glob(os.path.join(rag_qa_benchmarking_dir, "results",
"*.json"))

data = []
for file in json_files:
    with open(file) as f:
        json_data = json.load(f)
        json_data["qid"] = file.split("/")[-1].replace(".json", "")
        data.append(json_data)

results_df = pd.DataFrame(data).set_index("qid")
results_df["winner"].value_counts(normalize=True)
```

settings_tutorial

Setup

> This tutorial is available as a Jupyter notebook here ↗.

This tutorial aims to show how to use the `Settings` class to configure `PaperQA`.
Firstly, we will be using `OpenAI` and `Anthropic` models, so we need to set the `OPENAI_API_KEY` and `ANTHROPIC_API_KEY` environment variables.
We will use both models to make it clear when `paperqa` agent is using either one or the other.
We use `python-dotenv` to load the environment variables from a `.env` file.
Hence, our first step is to create a `.env` file and install the required packages.

```
# fmt: off
# Create .env file with OpenAI API and Anthropic API keys
# Replace <your-openai-api-key> and <your-anthropic-api-key> with your
actual API keys
!echo "OPENAI_API_KEY=<your-openai-api-key>" > .env # fmt: skip
!echo "ANTHROPIC_API_KEY=<your-anthropic-api-key>" >> .env # fmt: skip

!uv pip install -q nest-asyncio python-dotenv aiohttp fhlmi "paper-
qa[local]"
# fmt: on
```

```
import os

import aiohttp
import nest_asyncio
from dotenv import load_dotenv

nest_asyncio.apply()
load_dotenv(".env")
```

```
print("You have set the following environment variables:")
print(
    f"OPENAI_API_KEY:    {'is set' if os.environ['OPENAI_API_KEY'] else
'is not set'}"
)
print(
    f"ANTHROPIC_API_KEY: {'is set' if os.environ['ANTHROPIC_API_KEY']
else 'is not set'}"
)
```

We will use the `lmi` package to get the model names and the `.papers` directory to save documents we will use.

```
from lmi import CommonLLMNames

llm_openai = CommonLLMNames.OPENAI_TEST.value
llm_anthropic = CommonLLMNames.ANTHROPIC_TEST.value

# Create the `papers` directory if it doesn't exist
os.makedirs("papers", exist_ok=True)

# Download the paper from arXiv and save it to the `papers` directory
url = "https://arxiv.org/pdf/2407.01603"
async with aiohttp.ClientSession() as session, session.get(url,
timeout=60) as response:
    content = await response.read()
    with open("papers/2407.01603.pdf", "wb") as f:
        f.write(content)
```

The `Settings` class is used to configure the PaperQA settings.
Official documentation can be found here ↗ and the open source code can be found here ↗.

Here is a basic example of how to use the `Settings` class. We will be unnecessarily verbose for the sake of clarity. Please notice that most of the settings are optional and the defaults are good for most cases. Refer to the descriptions of each setting ↗ for more information.

Within this `Settings` object, I'd like to discuss specifically how the llms are configured and how `paperqa` looks for papers.

A common source of confusion is that multiple `llms` are used in paperqa. We have `llm`, `summary_llm`, `agent_llm`, and `embedding`. Hence, if `llm` is set to an `Anthropic` model, `summary_llm` and `agent_llm` will still require a `OPENAI_API_KEY`, since `OpenAI` models are the default.

Among the objects that use `llms` in `paperqa`, we have `llm`, `summary_llm`, `agent_llm`, and `embedding`:

- `llm`: Main LLM used by the agent to reason about the question, extract metadata from documents, etc.
- `summary_llm`: LLM used to summarize the papers.
- `agent_llm`: LLM used to answer questions and select tools.
- `embedding`: Embedding model used to embed the papers.

Let's see some examples around this concept. First, we define the settings with `llm` set to an `OpenAI` model. Please notice this is not an complete list of settings. But take your time to read through this `Settings` class and all customization that can be done.

```python
import pathlib

from paperqa.prompts import (
    CONTEXT_INNER_PROMPT,
    CONTEXT_OUTER_PROMPT,
    citation_prompt,
    default_system_prompt,
    env_reset_prompt,
    env_system_prompt,
    qa_prompt,
    select_paper_prompt,
    structured_citation_prompt,
    summary_json_prompt,
    summary_json_system_prompt,
    summary_prompt,
)
from paperqa.settings import (
    AgentSettings,
    AnswerSettings,
    IndexSettings,
    ParsingSettings,
    PromptSettings,
    Settings,
)

settings = Settings(
    llm=llm_openai,
    llm_config={
        "model_list": [
            {
                "model_name": llm_openai,
                "litellm_params": {
                    "model": llm_openai,
                    "temperature": 0.1,
                    "max_tokens": 4096,
                },
            }
        ],
        "rate_limit": {
            llm_openai: "30000 per 1 minute",
        },
    },
    summary_llm=llm_openai,
    summary_llm_config={
        "rate_limit": {
            llm_openai: "30000 per 1 minute",
        },
    },
```

```
    embedding="text-embedding-3-small",
    embedding_config={},
    temperature=0.1,
    batch_size=1,
    verbosity=1,
    manifest_file=None,
    paper_directory=pathlib.Path.cwd().joinpath("papers"),
    index_directory=pathlib.Path.cwd().joinpath("papers/index"),
    answer=AnswerSettings(
        evidence_k=10,
        evidence_detailed_citations=True,
        evidence_retrieval=True,
        evidence_summary_length="about 100 words",
        evidence_skip_summary=False,
        answer_max_sources=5,
        max_answer_attempts=None,
        answer_length="about 200 words, but can be longer",
        max_concurrent_requests=10,
    ),
    parsing=ParsingSettings(
        chunk_size=5000,
        overlap=250,
        citation_prompt=citation_prompt,
        structured_citation_prompt=structured_citation_prompt,
```

```
from paperqa import ask

response = ask(
    "What are the most relevant language models used for chemistry?",
settings=settings
)
```

```
        system=default_system_prompt,
        use_json=True,
        summary_json=summary_json_prompt,
        summary_json_system=summary_json_system_prompt,
        context_outer=CONTEXT_OUTER_PROMPT,
```

```
os.environ["OPENAI_API_KEY"] = ""
print("You have set the following environment variables:")
print(
    f"OPENAI_API_KEY:    {'is set' if os.environ['OPENAI_API_KEY'] else
'is not set'}"
)
print(
    f"ANTHROPIC_API_KEY: {'is set' if os.environ['ANTHROPIC_API_KEY']
else 'is not set'}"
)
```

```
                  }
```

```
response = ask(
    "What are the most relevant language models used for chemistry?",
settings=settings
)
```

```
        agent_prompt=env_reset_prompt,
        agent_system_prompt=env_system_prompt,
        search_count=8,
        index=IndexSettings(
            paper_directory=pathlib.Path.cwd().joinpath("papers"),

index_directory=pathlib.Path.cwd().joinpath("papers/index"),
        ),
    ),
)
```

```
settings.llm = llm_anthropic
settings.llm_config = {
    "model_list": [
        {
            "model_name": llm_anthropic,
            "litellm_params": {
                "model": llm_anthropic,
                "temperature": 0.1,
                "max_tokens": 512,
            },
        }
    ],
    "rate_limit": {
        llm_anthropic: "30000 per 1 minute",
    },
}
settings.summary_llm = llm_anthropic
settings.summary_llm_config = {
    "rate_limit": {
        llm_anthropic: "30000 per 1 minute",
    },
}
settings.agent = AgentSettings(
    agent_llm=llm_anthropic,
    agent_llm_config={
        "rate_limit": {
            llm_anthropic: "30000 per 1 minute",
        },
    },
    index=IndexSettings(
        paper_directory=pathlib.Path.cwd().joinpath("papers"),
        index_directory=pathlib.Path.cwd().joinpath("papers/index"),
    ),
)
settings.embedding = "st-multi-qa-MiniLM-L6-cos-v1"
response = ask(
    "What are the most relevant language models used for chemistry?",
settings=settings
)
```

Now the agent is able to use `Anthropic` models only and although we don't have a valid `OPENAI_API_KEY`, the question is answered because the agent will not use `OpenAI` models. See that we also changed the `embedding` because it was using `text-embedding-3-small` by default, which is a `OpenAI` model. `Paperqa` implements a few embedding models. Please refer to the documentation ↗ for more information.

Notice that we redefined `settings.agent.paper_directory` and `settings.agent.index` settings. `Paperqa` actually uses the setting from `settings.agent`. However, for convenience, we implemented an alias in `settings.paper_directory` and `settings.index_directory`.

In addition, notice that this is a very verbose example for the sake of clarity. We could have just set only the llms names and used default settings for the rest:

```
llm_anthropic_config = {
    "model_list": [{
            "model_name": llm_anthropic,
    }]
}

settings.llm = llm_anthropic
settings.llm_config = llm_anthropic_config
settings.summary_llm = llm_anthropic
settings.summary_llm_config = llm_anthropic_config
settings.agent = AgentSettings(
    agent_llm=llm_anthropic,
    agent_llm_config=llm_anthropic_config,
    index=IndexSettings(
        paper_directory=pathlib.Path.cwd().joinpath("papers"),
        index_directory=pathlib.Path.cwd().joinpath("papers/index"),
    ),
)
settings.embedding = "st-multi-qa-MiniLM-L6-cos-v1"
```

The output

`Paperqa` returns a `PQASession` object, which contains not only the answer but also all the information gatheres to answer the questions. We recommend printing the `PQASession` object (`print(response.session)`) to understand the information it contains. Let's check the `PQASession` object:

```
print(response.session)
```

```
print("Let's examine the PQASession object returned by paperqa:\n")

print(f"Status: {response.status.value}")

print("1. Question asked:")
print(f"{response.session.question}\n")

print("2. Answer provided:")
print(f"{response.session.answer}\n")
```

In addition to the answer, the `PQASession` object contains all the references and contexts used to generate the answer.

Because `paperqa` splits the documents into chunks, each chunk is a valid reference. You can see that it also references the page where the context was found.

```
print("3. References cited:")
print(f"{response.session.references}\n")
```

Lastly, `PQASession.session.contexts` contains the contexts used to generate the answer. Each context has a score, which is the similarity between the question and the context. `Paperqa` uses this score to choose what contexts is more relevant to answer the question.

```
print("4. Contexts used to generate the answer:")
print(
    "These are the relevant text passages that were retrieved and used to formulate the answer:"
)
for i, ctx in enumerate(response.session.contexts, 1):
    print(f"\nContext {i}:")
    print(f"Source: {ctx.text.name}")
    print(f"Content: {ctx.context}")
    print(f"Score: {ctx.score}")
```

Where to get papers

OpenReview

You can use papers from https://openreview.net/ ↗ as your database!
Here's a helper that fetches a list of all papers from a selected conference (like ICLR, ICML, NeurIPS), queries this list to find relevant papers using LLM, and downloads those relevant papers to a local directory which can be used with paper-qa on the next step. Install `openreview-py` with

```
pip install paper-qa[openreview]
```

and get your username and password from the website. You can put them into `.env` file under `OPENREVIEW_USERNAME` and `OPENREVIEW_PASSWORD` variables, or pass them in the code directly.

```
from paperqa import Settings
from paperqa.contrib.openreview_paper_helper import
OpenReviewPaperHelper

# these settings require gemini api key you can get from
https://aistudio.google.com/
# import os; os.environ["GEMINI_API_KEY"] = os.getenv("GEMINI_API_KEY")
# 1Mil context window helps to suggest papers. These settings are not
required, but useful for an initial setup.
settings = Settings.from_name("openreview")
helper = OpenReviewPaperHelper(settings,
venue_id="ICLR.cc/2025/Conference")
# if you don't know venue_id you can find it via
# helper.get_venues()

# Now we can query LLM to select relevant papers and download PDFs
question = "What is the progress on brain activity research?"

submissions = helper.fetch_relevant_papers(question)

# There's also a function that saves tokens by using openreview
metadata for citations
docs = await helper.aadd_docs(submissions)

# Now you can continue asking like in the [main tutorial]
(../../README.md)
session = await docs.aquery(question, settings=settings)
print(session.answer)
```

Zotero

It's been a while since we've tested this - so let us know if it runs into issues!

If you use Zotero ↗ to organize your personal bibliography,
you can use the `paperqa.contrib.ZoteroDB` to query papers from your library,
which relies on pyzotero ↗.

Install `pyzotero` via the `zotero` extra for this feature:

```
pip install paper-qa[zotero]
```

First, note that PaperQA2 parses the PDFs of papers to store in the database, so all relevant papers should have PDFs stored inside your database. You can get Zotero to automatically do this by highlighting the references you wish to retrieve, right clicking, and selecting *"Find Available PDFs"*. You can also manually drag-and-drop PDFs onto each reference.

To download papers, you need to get an API key for your account.

1. Get your library ID, and set it as the environment variable `ZOTERO_USER_ID`.
   - For personal libraries, this ID is given here ↗ at the part "*Your userID for use in API calls is XXXXXX*".
   - For group libraries, go to your group page `https://www.zotero.org/groups/groupname`, and hover over the settings link. The ID is the integer after /groups/. (*h/t pyzotero!*)
2. Create a new API key here ↗ and set it as the environment variable `ZOTERO_API_KEY`.
   - The key will need read access to the library.

With this, we can download papers from our library and add them to PaperQA2:

```
from paperqa import Docs
from paperqa.contrib import ZoteroDB

docs = Docs()
zotero = ZoteroDB(library_type="user")  # "group" if group library

for item in zotero.iterate(limit=20):
    if item.num_pages > 30:
        continue  # skip long papers
    await docs.aadd(item.pdf, docname=item.key)
```

which will download the first 20 papers in your Zotero database and add them to the `Docs` object.

We can also do specific queries of our Zotero library and iterate over the results:

```
for item in zotero.iterate(
    q="large language models",
    qmode="everything",
    sort="date",
    direction="desc",
    limit=100,
):
    print("Adding", item.title)
    await docs.aadd(item.pdf, docname=item.key)
```

You can read more about the search syntax by typing `zotero.iterate?` in IPython.

Paper Scraper

If you want to search for papers outside of your own collection, I've found an unrelated project called [paper-scraper ↗] that looks
like it might help. But beware, this project looks like it uses some scraping tools that may violate publisher's rights or be in a gray area of legality.

```
from paperqa import Docs

keyword_search = "bispecific antibody manufacture"
papers = paperscraper.search_papers(keyword_search)
docs = Docs()
for path, data in papers.items():
    try:
        await docs.aadd(path)
    except ValueError as e:
        # sometimes this happens if PDFs aren't downloaded or readable
        print("Could not read", path, e)
session = await docs.aquery(
    "What manufacturing challenges are unique to bispecific
antibodies?"
)
print(session)
```

tests

stub_data

Gravity hill

> "Magnetic hill" and "Mystery hill" redirect here. For other uses, see Magnetic Hill (disambiguation) ↗ and Mystery Hill (disambiguation) ↗.

A **gravity hill**, also known as a **magnetic hill**, **mystery hill**, **mystery spot**, **gravity road**, or **anti-gravity hill**, is a place where the layout of the surrounding land produces an illusion ↗, making a slight downhill slope appear to be an uphill slope. Thus, a car left out of gear will appear to be rolling uphill against gravity ↗.

Although the slope of gravity hills is an illusion, sites are often accompanied by claims that magnetic or supernatural forces are at work. The most important factor contributing to the illusion is a completely or mostly obstructed horizon. Without a horizon, it becomes difficult for a person to judge the slope of a surface, as a reliable reference point is missing, and misleading visual cues can adversely affect the sense of balance. Objects which one would normally assume to be more or less perpendicular to the ground, such as trees, may be leaning, offsetting the visual reference.

A 2003 study looked into how the absence of a horizon can skew the perspective on gravity hills, by recreating a number of antigravity places in the lab to see how volunteers would react. In conclusion, researchers from the Universities of Padua and Pavia in Italy found that without a true horizon in sight, the human brain could be tricked by common landmarks such as trees and signs.

The illusion is similar to the Ames room ↗, in which objects can also appear to roll against gravity.

The opposite phenomenon—an uphill road that appears flat—is known in bicycle racing ↗ as a "false flat" ↗.

See also

- List of gravity hills ↗
- The Crooked House ↗ – a pub (now demolished) with an internal gravity hill illusion.

References

External links