Surveying and building an Automatic Knowledge Agent

Javier Diego Fernandez

January 11, 2026

1 How is using Ai2 Paper Finder different from Google Scholar?

Although the internals of the Google Scholar (GS) search algorithm are not public and we therefore can't do a low-level technical comparison between it and the Ai2 Paper Finder (Ai2PF) tool, we can glean their differences by comparing their recommendations for some queries. In the next subsection, we will show some examples of the top recommendation made by each of the systems. With them, we will try to infer underlying differences in how the two systems work.

1.1 Better recommendations for high-level queries

Q: "most significant papers in the early era of neural networks"

- Ai2PF: Learning representations by back-propagating errors [1].
- GS: Evolutionary design of neural network architectures: a review of three decades of research [2].

Most people would agree that the recommendation made by the Ai2PF is better than the one made by Google Scholar. The query is asking for papers in the early era, and the GS recommendation is returning a paper from 2008. It is interesting to look at the text highlighted by GS for the match:

> The **first period** covers **initial** attempts to evolve simple ANN ... We aimed to include all **relevant papers** without any selection ... The rest of this review **paper** is organized as follows: In Sec. 2 ...

As we can see, the tool is probably trying to exploit some notion of synonymy between the expressions "early era" in the query and "first period" in the text of the paper, and between "significant papers" in the query and "relevant papers" in the results. Although this may sound sensible, this approach fails because GS is trying to find the expressions directly in the text of the papers. Ai2PF is able to recognize that the query is asking for papers whose theme is neural networks, even if the title or the paper contents do not contain the expression directly. A similar thing is happening with the "early era" expression, where the Ai2PF is able to recognize that the paper should be old, even if the paper itself does not mention the word "early" or a synonym directly in the text.

1.2 Comparison Conclusion

Appendix A shows two more comparisons of queries, all of which paint a similar picture: Google Scholar is relying on keyword matching in the paper title and body, even if trying to match low-level synonyms. The Ai2PF tool, on the other hand, is able to capture a higher-level understanding of the meaning of the query. A significant difference is that the Ai2PF relies on the more modern technique of using text embeddings to match papers and snippets, together with higher-level checks using LLMs, whereas GS still relies on a more classical approach to search based on keyword matching. This difference will be clearer once we review the internal architecture of the Ai2PF tool.

2 How does Ai2 Paper Finder work internally?

Fortunately, the Ai2 Paper Finder is open-source [1], so we can read its source code to understand how it functions. Figure 1 shows a high-level overview of the architecture of the Ai2PF. The workflow from query to response can be broadly divided into three phases. We will give a high-level overview of each of them in the following subsections.

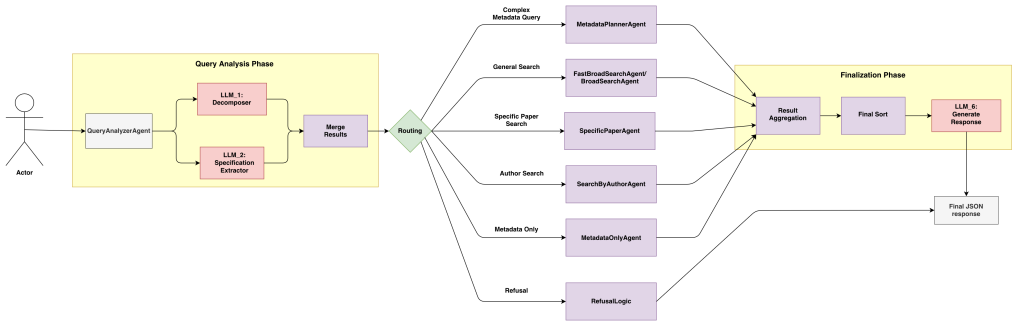

Figure 1: General architecture of the Allen Institute Paper Finder Tool

2.1 Query analysis and routing phase

Immediately after receiving the call from the client, the query sent by the user is passed to the `query_analyzer.py` file. The file execute two sets of asynchronous LLM calls. These calls try to classify the intent and structure of the query so that it can be routed to the appropriate agent further down the chain:

- Decomposer LLM: This block corresponds to a series of parallel calls done to the same LLM, with each one designed to extract a different and specific aspect of the query. One call, for example, is designed to extract exclusively the author names, and includes a prompt asking the model to "identify the authors whose papers are being requested ...". Another one tries to identify if the query is asking for highly influential papers and includes the prompt "... decide whether the query asks for central papers or less

[1] The source code for the Allen Insitute Paper Finder tool is available at the following Github repo: https://github.com/allenai/asta-paper-finder

cited papers ...". In total, there are 11 different calls done to the model, each extracting different specific information. They can be explored at the `query_analyzer_prompts.py` file.

- Specification Extractor LLM: This LLM is designed to handle complex queries that may involve boolean connectives and aggregations (e.g. "Papers released by either author A or author B, between 1990 and 1993 and in either journal C or journal D"). The prompt also deals with citations, allowing queries that talk indirectly about a paper (e.g. "Find papers that were cited as inspiration by the transfomers paper"), among other things. The LLM returns a complex JSON with all the fields extracted and the logical connectives attached to them, the prompt used can be inspected in the `extract_specifications.md` file.

2.2 Paper retrieval agents

After calling the previous two models, the system uses the structured output returned by them to decide which "agent" to route the query to. The underlying motivation behind the use of several agents seems to be related to performance and cost. There are simpler agents that mostly just query the Ai2 APIs, and complex agents that use LLMs to suggest papers and do relevance evaluations. A complex LLM may not be necessary for every single query, so we can save time and cost by using a simpler model if the system doesn't require a more intelligent analysis. The agents that can be used are the following ones:

1. `MetadataPlannerAgent`: If the system determines the query to be "complex" by having logical criteria in it or indirect references, the JSON string returned by the specification extractor LLM is used to obtain the papers. The agent reads the JSON specification, queries the Ai2 API, and executes a small logic agent that implements the operations defined in it to obtain the papers. The agent is reminiscent of a very simplified SQL query engine and can be inspected in the `ops.py` file

2. `FastBroadSearchAgent/BroadSearchAgent`: These agents are triggered when the user specifies a broad criteria for the search but no specific title or paper. The system executes a mixture of keyword searches, vector database searches, and citation retrievals from the Ai2 API to create a large pool of candidates. These candidates are then filtered for revelance to the query with the use of a reranker, specifically Cohere Rerank 3. Finally, the candidates are passed to an LLM judge that gives a final evaluation of the relevance of each paper.

3. `SpecificPaperByTitleAgent`: If the decomposer LLM determines that the query asked for a specific title (e.g. "Attention is All You Need"), the query is routed to this agent. It does a very straightforward search in the semantic scholar API to fetch the paper using the title. The code can be inspected at the `specific_paper_by_title_agent.py` file.

4. `SpecificPaperByNameAgent`: Triggered when the user query is asking for a single paper, but not by its explicit title (e.g. "The transformers paper" instead of "Attention is All You Need"). This agent uses an LLM to hallucinate possible names for the paper, it downloads candidate papers from the Ai2 database, and combines them with deterministic keyword searches in the paper body to try to find the best match. It then uses a deterministic scorer to decide the best paper to keep.

5. `SearchByAuthorsAgent`: This agent is triggered when the query asks for titles coming from specific authors. Using the possible authors coming from the query, it queries the semantic scholar API for possible author names and ranks them. Once the author has been identified, the system tries to find the paper by running a keyword search and having an LLM suggest its title. Finally, all the candidate papers are evaluated by an LLM and the best match is returned.

6. `MetadataOnlySearchAgent`: This agent is triggered when only metadata like year or venue are specified for the paper, but no author, paper title, or broad topic that would trigger the previously described agents. The system takes whatever metadata input was specified and makes a straightforward query to the Ai2 API to fetch the papers.

2.3 Result aggregation and final response

Each of the specific agents returns a list of candidate papers to be displayed to the user and a final for sorting is done. The sorting criteria depends on several factors that include the recency of the paper, which prioritizes newer works, the influence of the paper, prioritizing papers with more citations, and an LLM evaluation and rerank in case the agent used an LLM to obtain the papers (like the `BroadSearchAgents` and `SearchByAuthorsAgents`). The last step is sending the ranked results to an LLM to craft the last response that will be displayed to the user. The response depends on the specific agent used, but includes information about what results the system found, what the system searched for exactly in the API, and a suggestion for the next actions for the user to try. The possible templates that can be displayed to the user are shown in the `explain.toml` file.

3 One task inside Ai2 Paper Finder: Generating snippet embeddings

The description made in the previous section was only a very general overview of the internal workings of the system. Inside of it, and specially inside the `FastBroadSearchAgent` and `BroadSearchAgent`, there are several subparts that are the current focus of the scientific research. To give one example, both broad search agents use a "snowball agent" designed to retrieve papers that initial candidates cite, or that are cited by the candidates. Recent research has moved from these straightforward one-hop retrievals to more sophisticated systems that rely on building graphs. At query time, these systems rely on learnable mechanisms to prune the graph, retaining only the nodes that are contextually helpful to answer the query [3]. There are many more examples of recent literature that could improve the performance of the current implementation of the Ai2PF.

3.1 Snippet embeddings in the Ai2PF

For this assignment, I decided to focus on a foundational task needed to build RAG systems: the construction of a vector embedding database that can be used to find relevant papers for the query. In the Ai2PF, a vector embedding database is used by both the `FastBroadSearchAgent` and `BroadSearchAgents` to find snippets extracted from papers that match the content of the original query. These snippets serve two purposes:

1. The papers that contain the snippets are considered as possible candidates to be displayed to the user.

2. If the snippets contain citations, the papers that are cited in them are also considered as candidates.

As already mentioned, these candidates are collected and filtered with a reranker and an LLM judge before returning the final papers to the user.

3.2 Overview of the system

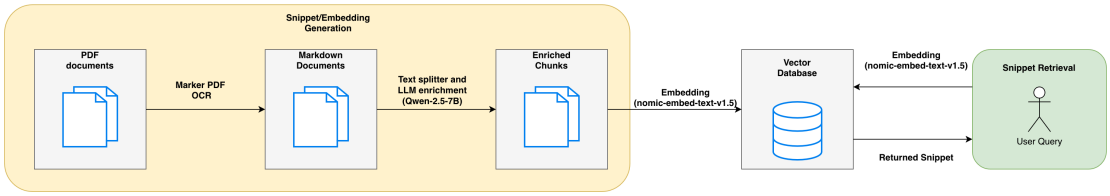

Figure 2: Architecture of snippet generation and retrieval

Figure 2 shows an overview of the system built. It consists of two main stages: snippet generation and snippet retrieval. In the snippet generation phase, the original papers in PDF format are transformed to text snippets and stored in a vector database. This is a one-off process only done when adding new papers to the database. In the retrieval stage, the user query is transformed into an embedding, and matched to the closest N snippets in the databases, which are then returned to the user.

3.3 Snippet generation

The generation phase executes the following actions sequentially:

1. *Markdown OCR*: The first step is to convert the PDF files to a text format for which embeddings can be generated and stored. For this, I relied on the Marker tool [4]. Marker consists of a mixture of OCR, layout analysis, and heuristic algorithms that work together to understand the structure of the document and convert it to a variety of formats. For this project, they where transformed to markdown format with only the most important structural elements, like headers, equations, and tables, preserved.

2. *Snippet Generation*: Text embedding models are limited in the size of the context window they can process and adequately compress. For this reason, it is necessary to break up the papers into chunks before embedding them. To do this, I relied on the straightforward `langchain_text_splitters` [5] Python library.

3. *Snippet Enrichment*: Creating embeddings directly from the snippet like the Ai2 API does can have an important downside related to context loss. The specific snippet extracted could lack the overall context of what is it talking about, since that can generally be mentioned in other chunks. This in turn can create worse matches at

query time that return less relevant snippets. For this reason, a new technique called *contextual retrieval* [6] has been suggested. This technique works by simply adding an LLM-generated (in this case using `Qwen2.5-7B-Instruct`[7]) prefix to the chunk that explains what it is discussing. This prefix is then prepended to the chunk before generating the embeddings.

4. *Embedding*: With the context added, we can proceed to generate the embeddings and store them in the vector database for later retrieval. To generate the embeddings, I used the `nomic-embed-text-v1.5`[8] model. This model is based on the BERT architecture with significant modifications and trained in a complex pipeline that includes language modeling, vast unsupervised training with weak pairs, and supervised fine-tuning with high-quality labeled data. The model achieves comparable performance to commercial solutions.

5. *Storage*: With the embeddings on the enriched chunk, we can proceed to store the chunks in the database. For simplicity, I used ChromaDB [9]. ChromaDB is a lightweight and local vector embedding database that uses SQLite as its backend engine.

3.4 Snippet retrieval

For retrieval, we similarly embed the user query with the `nomic-embed-text-v1.5` embedding model, and use ChromaDB to obtain the closest three chunks.

3.5 Results

The resulting system can be studied openly at its Github repository [2]. The whole execution, from the paper PDFs to the query retrieval can be inspected in the `runner.ipynb` notebook. For a given query, the system returns the following three elements for every matching chunk:

- The source paper that contains the chunk.
- The LLM-generated context prefix.
- The matched chunk.
- The ids of the cited papers in the chunk, if there are any.

Compared to the Semantic Scholar snippet API [3] the system provides two main advantages:

1. The improved OCR generation correctly preserves titles, equations, and tables that the Semantic Scholar API misses. Appendix B shows a comparison of the snippets returned by each system for the same paper section. The correct mathematical notation preserves the meaning of the mathematical notation that is lost in the original API.

2. As the Anthropic technical blogpost shows [6], the use of contextual embeddings reduces the rate of failed retrievals in RAG systems.

This demo is just a first encounter to the methods and tools present in the modern RAG ecosystem. I look forward to working on state-of-the-art approaches to RAG like RAG-Reasoning [10, 11], GraphRAG [12, 13], and multimodal RAG [14].

[2] Code available at: `https://github.com/javierdiegof/paper_embedding`

[3] Endpoint URL available at: `https://api.semanticscholar.org/graph/v1/snippet/search`

References

[1] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[2] Dario Floreano, Peter Dürr, and Claudio Mattiussi. Evolutionary design of neural network architectures: A review of three decades of research. *Artificial Life*, 14(1):47–79, 2008.

[3] Yuntong Hu, Zhihan Lei, Zheng Zhang, Bo Pan, Chen Ling, and Liang Zhao. Grag: Graph retrieval-augmented generation, 2024.

[4] Vik Paruchuri. Marker: Convert pdf to markdown + json quickly with high accuracy. `https://github.com/datalab-to/marker`, 2023.

[5] LangChain. Text splitters - docs by langchain. `https://docs.langchain.com/oss/python/integrations/splitters`, 2025. Accessed: 2025-12-27.

[6] Anthropic. Introducing contextual retrieval. `https://www.anthropic.com/engineering/contextual-retrieval`, September 2024. Accessed: 2025-12-27.

[7] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2.5 technical report, 2024.

[8] Zach Nussbaum, John X. Morris, Brandon Duderstadt, and Andriy Mulyar. Nomic embed: Training a reproducible long context text embedder, 2024.

[9] Chroma Core. Chroma: Open-source search and retrieval database for ai applications, 2025. Accessed: 2025-12-27.

[10] Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. Search-r1: Training llms to reason and leverage search engines with reinforcement learning. *arXiv preprint arXiv:2503.09516*, 2025.

[11] Yangning Li, Weizhi Zhang, Yuyao Yang, Wei-Chieh Huang, Yaozu Wu, Junyu Luo, Yuanchen Bei, Henry Peng Zou, Xiao Luo, Yusheng Zhao, et al. Towards agentic rag with deep reasoning: A survey of rag-reasoning systems in llms. *arXiv preprint arXiv:2507.09477*, 2025.

[12] Mariam Barry, Gaetan Caillaut, Pierre Halftermeyer, Raheel Qader, Mehdi Mouayad, Fabrice Le Deit, Dimitri Cariolaro, and Joseph Gesnouin. Graphrag: Leveraging graph-based efficiency to minimize hallucinations in llm-driven rag for finance data. In *Proceedings of the 1st Workshop on Generative AI and Knowledge (GenAIK) at COLING 2025*. Association for Computational Linguistics, January 2025.

[13] Tianyang Xu, Haojie Zheng, Chengze Li, Haoxiang Chen, Yixin Liu, Ruoxi Chen, and Lichao Sun. Noderag: Structuring graph-based rag with heterogeneous nodes. *arXiv preprint arXiv:2504.11544*, 2025.

[14] Vishesh Tripathi, Tanmay Odapally, Indraneel Das, Uday Allu, and Biddwan Ahmed. Vision-guided chunking is all you need: Enhancing rag with multimodal document understanding. *arXiv preprint arXiv:2506.16035*, 2025.

[15] Alonzo Church. *The Calculi of Lambda-Conversion*, volume 6 of *Annals of Mathematics Studies*. Princeton University Press, Princeton, NJ, 1941.

[16] Jonathan P Bowen. The impact of alan turing: Formal methods and beyond. In *Engineering Trustworthy Software Systems*, pages 202–235. Springer, 2019.

[17] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, volume 35, pages 16344–16359, 2022.

[18] Adrián Hernández and José M Amigó. Attention mechanisms and their applications to complex systems. *Entropy*, 23(3):283, 2021.

Appendix A Queries

A.1 Ability with indirect queries

Q: "The paper where Alan Turing's doctoral advisor introduced his formal calculus."

- Ai2PF: The calculi of lambda-conversion [15].
- GS: The impact of Alan Turing: formal methods and beyond [16].

Again, we're able to see that the Ai2PF tool is able to identify that the author we're looking for is Alonzo Church and not Alan Turing. In a similar manner to the previous example, Google Scholar is trying to match the words in the query to what is inside the paper. Moving on to a more realistic example of what a practicing researcher would ask ...

A.2 Better topic recognition

Q: "Breakthrough papers trying to reduce the complexity in the attention mechanism"

- Ai2PF: FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awarenes [17].
- GS: Attention mechanisms and their applications to complex systems [18].

Again, we can observe that GS is trying to match the word "complexity" in the query with the word "complex" in the title of the paper, without realizing that they refer to different topics altogether.

Appendix B Snippet Comparison

For the query:

> The description of the initialization bias correction equation for Adam

The Ai2PF API returns the following snippet:

> ... Let us initialize the exponential moving average as v 0 = 0 (a vector of zeros). First note that the update at timestep t of the exponential moving average
>
> where g 2 t indicates the elementwise square g t g t) can be written as a function of the gradients at all previous timesteps:
>
> We wish to know how E[v t], the expected value of the exponential moving average at timestep t, relates to the true second moment E[g 2 t], so we can correct for the discrepancy between the two.

The new encoder returns a much clearer reproduction of the paragraph. With correct markdown annotations for section titles and correct syntax for equations:

3 INITIALIZATION BIAS CORRECTION

... Let us initialize the exponential moving average as $v_0 = 0$ (a vector of zeros). First note that the update at timestep t of the exponential moving average $v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (where g_t^2 indicates the elementwise square $g_t \odot g_t$) can be written as a function of the gradients at all previous timesteps:

$$v_t = (1 - \beta_2) \sum_{i=1}^{t} \beta_2^{t-i} \cdot g_i^2 \tag{1}$$

We wish to know how $\mathbb{E}[v_t]$, the expected value of the exponential moving average at timestep t, relates to the true second moment $\mathbb{E}[g_t^2]$, so we can correct for the discrepancy between the two.

The new encoder adds a prefix explanation to the snippet indicating what it is talking about. The explanation is added to the snippet before it is stored in the database. This additional context increases the likelihood of finding a relevant match when searching a query:

```
The text discusses the derivation of the initialization bias correction
term for the second moment estimate in the Adam optimization algorithm,
explaining how the exponential moving average of the squared gradient is
updated over time and how this process corrects for initialization bias.
```