**Slide 1 — Academic Research Online Agent**

Hi everyone and welcome. My name is Rayyan and today I will be walking you through a system I have developed called the Academic Research Online Agent. Basically, this is a tool built for the backend that helps collect, summarize, and neatly save academic content from websites. It’s designed to fix common problems researchers face when doing literature reviews, especially the boring and time-consuming task of copying and pasting information by hand. Instead of doing everything manually, the system turns those steps into an automated process that can be used through a simple web interface (API). It also saves useful details like the original web address and the time the content was collected, so everything can be checked and trusted later.

**Slide 2 — Objectives & Value Proposition**

This project is built around three clear goals, two practical and one structural: First, on the practical side, it automatically finds and pulls academic content from the web, saving researchers time in the early stages of research. Second, it creates short, easy-to-search summaries and useful keywords using TF-IDF. This helps users quickly decide if a source is worth reading. Third, on the structural side, the system is built to be repeatable and easy to improve with built-in tests and clear documentation so, others can use or extend it with confidence. These goals shaped important design choices like using clear, predictable methods instead of black-box models, building a simple and clean API, and packaging the system as a single-file SQLite setup, which works well for small research teams.

**Slide 3 — System Workflow Overview**

Let’s now look at the big picture, the system’s **end-to-end workflow**. It follows a simple pipeline: **crawl → process → store → serve**. How it works?

A user sends a POST request with a URL to the API.

The crawler fetches and cleans up the page content.

The processor summarizes the content and extracts keywords using TF-IDF.

The results are stored.

The API then makes them retrievable via GET or exportable into CSV.

What’s useful about this separation is modularity. For example, if in the future we decide to replace the TF-IDF summarizer with a transformer-based model, that change can be made inside the processing module **without affecting the rest** of the pipeline.

**Slide 4 — Core Components of the Agent**

There are four core components**:**

**Crawler**: Built using requests and BeautifulSoup, the crawler gets the HTML from a webpage, handles timeouts and retries, and uses smart rules to pull out the main article while skipping things like headers, menus, and footers. It also follows robots.txt rules and slows down if needed to avoid overloading websites.

**Processor:** This part uses scikit-learn’s TF-IDF tool. It breaks the text into sentences, removes symbols like commas and periods, filters out common words, and then scores the sentences by importance. The top ones are picked to create a short summary. Keywords are chosen based on how often and how meaningfully they appear with duplicates and word endings cleaned up.

**Storage Layer**: A simple one-file SQLite database is used here, managed with SQLAlchemy. It safely saves data and keeps it even after the system is turned off, so users don’t need a big database setup to get started.

**API Layer**: Built with Flask, this part provides tools to process new documents, list saved ones, and export the results. Each part of the system is tested to make sure it works properly and can be improved easily later on.

**Slide 5 — End-to-End Demo Flow**

Let’s walk through what an actual run looks like.

Step one: send POST /process with the target URL.

Step two: the crawler retrieves the document and extracts readable text and metadata such as the title and meta-description when available.

Step three: the processor applies cleaning, stopword removal, TF-IDF scoring, and selects top-ranked sentences as an extractive summary while collecting keyword candidates and deduplicating them.

Step four: results are stored via SQLAlchemy to SQLite with atomic transactions to avoid partial writes.

Step five: the API returns a JSON object confirming the insert, including row id.

Users can then call GET /documents to list all entries or run the export\_to\_csv.py script to generate a CSV for further analysis. The system also handles edge cases, like very short pages or low-text pages, using fallback heuristics to still extract useful information.

**Slide 6 — Implementation & Code Structure**

In building this system, we focused on making it clear, consistent, and easy to work with. We used Flask to keep things lightweight and simple to document. For summarizing content, TF-IDF gives us results that are both predictable and easy to explain. We chose SQLite to avoid the hassle of complex database setup, and SQLAlchemy makes it easy to upgrade the storage system later if needed. To get web content, we rely on requests and BeautifulSoup, which work well for parsing regular, static web pages. Now, when it comes to speed, the system is built in a modular way so we can batch URLs, use multiple CPU cores for crawling, or even move processing to background workers like Celery, all without changing how the API behaves. We also added structured logs and clean JSON responses to support automated monitoring and alerting which is great for scaling or team-based use. And finally, TF-IDF settings like ngram\_range, max\_df, and min\_df can be adjusted. This gives us control to filter out noise or focus on key phrases, depending on the use case.

**Slide 7 — System Setup & Execution**

To run the system, the steps are straightforward. First, you create and activate a virtual environment, this keeps the project’s dependencies separate from the rest of your system. Then, install everything you need using pip. After that, you can run pytest to make sure all parts of the system are working correctly. Once that’s done, you start the Flask API, and you can send test URLs to the /process endpoint to see it in action. The exact commands for all of this are listed in the README, including tips for Windows users, especially if you're using PowerShell When you’re ready to export your results, just run the included script, and it will generate a CSV file called documents.csv. You can open this in Excel or any spreadsheet app to explore the data further. And if you want to check that everything is running correctly, the README also includes sample URLs and what the summaries should look like, so it’s easy to double-check that everything works as expected.

**Slide 8 — Activate Virtual Environment**

Here, we’re showing the screenshot taken right after setting up the virtual environment and installing the project dependencies. This step is important because it keeps the project’s packages separate from the rest of the system, avoiding conflicts. The screenshot shows the exact commands used, so new users can follow along easily. We recommend saving this kind of setup proof in the project handover, it really helps future maintainers.

**Slide 9 — Install Requirements**

This screenshot shows what it looks like when we run pip install with our requirements.txt file. It confirms that all the necessary packages are downloaded and installed properly. We pinned the package versions carefully to prevent installation errors or version mismatches which is a common problem when setting up Python projects. This also helps identify any slowdowns due to package mirrors or network issues.

**Slide 10 — Run Tests**

Next, we have the output from running our test suite using pytest. You can see that unit tests passed successfully, confirming that the core parts, the crawler, processor, and storage are working as expected. We’ve also added some simple tests for the API just to make sure it responds correctly. These tests focus on predictable input and output, while our integration tests check that everything works smoothly together. The screenshot includes useful details like timing and assertion messages, which are helpful for debugging or improving coverage.

**Slide 11 — Start API**

This screenshot shows the Flask server running and listening on the expected port. It also logs the incoming POST requests to /process, confirming that the API is active and receiving input. These logs are especially useful for spotting any delays or unexpected behavior. In a team setup, we recommend collecting these logs centrally to make debugging easier across the team.

**Slide 12 — Process URL**

Here, we show what happens when you send a URL to the API. The screenshot captures a real response in JSON which includes the article’s title, a summary, keywords, and a timestamp. It also returns a unique row ID, so you can trace the data back in the database. We have tested this against sample articles and confirmed that the summaries match up well with key ideas selected by humans, which means the TF-IDF summarizer is doing a solid job.

**Slide 13 — Export CSV**

Finally, this screenshot shows the exported CSV file opened in a spreadsheet app. Each row includes fields like the ID, title, URL, summary, keywords, and the time it was added. This makes it easy to filter and sort results, or to import them into reference managers or other research tools. We kept the format simple on purpose — to make it easy to use in just about any workflow.

**Slide 14 — Testing Results & Verification**

For testing and verification, we took a thorough approach. We started with **unit tests** to check each component in isolation, then used **functional tests with curl** to make sure the API behaved as expected. We also did **manual checks** on the database to confirm that records were being saved correctly. Our key checks included:

Making sure all database fields matched the expected structure,

Verifying that a POST request successfully inserted a new record, and

Ensuring the export to CSV worked as intended.

The tests were written to focus on **overall behavior**, rather than exact string matches, which makes them more reliable across different environments. We also noted and documented known **edge cases and failure modes**, along with tips for fixing them. On the performance side, we tracked how long it takes to process a typical URL and how much memory is used and the system handled most pages in just a few seconds.

**Slide 15 — System Outputs (Evidence)**

This slide highlights what the system produces after processing. You get: A structured JSON response from the /process endpoint, A SQLite database entry saved under /documents, An exported CSV file, and Flask logs showing exactly what happened and when. These outputs make it easy to track and debug issues. If something fails, you can trace it from the logs to the database to the CSV which is very useful for audits or troubleshooting. We have included example log files in the project evidence folder.

**Slide 16 — Validation & Quality Assurance**

For QA, we took a multi-layered approach. We ran unit tests with pytest, performed end-to-end tests across the whole pipeline, and manually checked the database and CSV rows to confirm the content looks right. We also sampled processed documents to review the clarity of the summaries and how relevant the keywords were. And we made sure important fields like timestamps and source URLs were always included to keep things traceable for academic work.

**Slide 17 — Challenges, Fixes & Technical Choices**

Like most real-world projects, we faced some technical hurdles along the way. Library compatibility issues, especially with scikit-learn and SQLAlchemy on newer Python versions, led to version mismatches and warnings during early installs. SQLite locking problems showed up when more than one process tried to write to the database at the same time. This is a known limitation of file-based databases. We also had to fine-tune the crawler’s retry logic and timeouts to handle unreliable pages and avoid getting blocked by sites.

To fix these: We pinned dependency versions in requirements.txt to ensure everyone runs the same working stack. We upgraded SQLAlchemy to avoid deprecation issues and reduce warning noise. For SQLite, we introduced a single-writer policy, and we recommend switching to PostgreSQL for teams needing concurrent access. PostgreSQL supports connection pooling, better handling of parallel writes, and easier scaling. These choices helped us strike a balance between simplicity for individuals and scalability for teams.

**Slide 18 — Implementation vs. Original Design Proposal**

The system we built stays true to the original proposal in both architecture and goals. The idea was always to separate crawling, processing, storage, and serving into modular layers and that’s exactly what we implemented. We made one major design trade-off: we chose TF-IDF summarization over large language models or neural approaches. Why? Because TF-IDF is fast, explainable, and easy to reproduce across machines which fits well with academic environments where reproducibility is a priority. Everything is modular. That means you can upgrade parts, swap in a smarter summarizer, add JavaScript rendering for dynamic content, or connect to a cloud database without rewriting the system or breaking the API. The REST interface stays consistent, even as internals change, making it easier for others to adopt or contribute to the project over time.

**Slide 19 — References**

This system is built on widely used, well-documented libraries and frameworks. Our references include:

* Python Software Foundation (2024) Python 3.13 Documentation. Available at: https://docs.python.org/3.13/ (Accessed: 5 October 2025).
* Pallets Projects (2024) Flask Documentation. Available at: https://flask.palletsprojects.com/en/latest/ (Accessed: 5 October 2025).
* SQLAlchemy (2024) SQLAlchemy Documentation. Available at: https://docs.sqlalchemy.org/en/20/ (Accessed: 5 October 2025).
* Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M. and Duchesnay, É. (2011) Scikit-learn: Machine Learning in Python, Journal of Machine Learning Research, 12, pp. 2825–2830. Available at: https://jmlr.org/papers/volume12/pedregosa11a/pedregosa11a.pdf (Accessed: 5 October 2025).
* Manning, C.D., Raghavan, P. and Schütze, H. (2008) Introduction to Information Retrieval. Cambridge: Cambridge University Press. Available at: https://nlp.stanford.edu/IR-book/ (Accessed: 5 October 2025).
* Richardson, L. and Ruby, S. (2007) RESTful Web Services. Sebastopol, CA: O’Reilly Media. Available at: https://www.oreilly.com/library/view/restful-web-services/9780596529260/ (Accessed: 5 October 2025).
* Richardson, L. (2004) Beautiful Soup Documentation. Available at: https://www.crummy.com/software/BeautifulSoup/bs4/doc/ (Accessed: 5 October 2025).

**Slide 20 — Conclusion**

Finally, this system isn’t just a scraper or summarizer, it’s a practical tool for academic workflows. It turns **repetitive, manual steps** into a clean, automated process that saves time and improves consistency. The modular architecture, tested components, and REST API make it **easy to run, easy to extend**, and **easy to trust**. Every processed document is saved with its source, time of creation, summary, and keywords which supporting reproducibility and traceability, which are vital in research. We see this project not as a finished product, but as a **foundation** others can build on whether that means plugging in a better summarizer, scaling to more users, or adding more export formats. Thank you for your attention.