EECS E6893 Big Data Analytics - Fall 2026

Homework Assignment 5: Big Data Systems for AI Applications

Due Thursday, November 19, 2026, 11:59 PM ET.

Total: 100 points

Assignment 5 Guidelines

An LLM application is only as good as the data system around the model. Before a model sees a question, the system must ingest and clean its sources, keep them current, protect personal data, retrieve the right evidence and decide whether that evidence is enough. After it answers, the system must cite its sources, keep working when the model is slow or failing, and measure whether any of this works.

The assignment has two standard questions on a provided dataset, and a mini-project you do in addition to them. The dataset is the help center of a fictional outdoor retailer, Northbeam Outfitters, whose support team wants an assistant that answers customer questions. Q1 builds the ingestion path of that assistant and Q2 builds its query path. The Mini-Project (Q3) asks you to design and prototype the same kind of system for a use case and dataset of your own.

Question	Topic	Points
Q1	Knowledge ingestion pipeline (standard task)	30
Q2	Grounded retrieval and production patterns (standard task)	30
Q3	Mini-Project: your own AI data system (additional part)	40
Total		**100**

Complete this assignment individually. You may reuse your own code from earlier homework. Cite adapted code, models, libraries and data sources, and follow the course academic-integrity policy. Background reading is listed under References at the end.

Data and Setup

Download `hw5_starter.zip` from CourseWorks (Files/HW5). Run `HW5_Starter_Guide.ipynb` first: it tours the data, explains the evaluation code and runs a baseline. Then complete Q1 and Q2 in `HW5_Student_TODO.ipynb`. A free Google Colab CPU runtime is enough; no GPU, paid API or API key is required. Spark runs in local mode inside Colab; Dataproc is also acceptable.

File	Contents
`kb_articles.jsonl`	Help-center articles, one row per version: `article_id`, `version`, `title`, `category`, `body`, `effective_date`, `status` (active, retired, draft), `source` (official, community). Contains malformed and duplicate records on purpose.
`tickets.csv`	2,400 resolved support tickets. They contain synthetic personal data.
`kb_updates.jsonl`	Article change events, in the order they arrive.
`golden_dev.jsonl`	66 labelled questions: an `as_of` date, a question type, the current gold documents and a short answer fact.
`query_log.csv`	600 customer queries in two phases, for the caching experiment.
`pii_sample.csv`	200 sentences with labelled personal-data spans.
`hw5lib.py`	Provided loaders, baseline, evaluation functions, failure simulator and timing helpers. Do not edit.
`hw5_system.py`	Your final system's entry point. It starts as the baseline.

All Northbeam data is synthetic. It does not describe a real company, policy or customer.

Evaluation

Chunk identifiers have the form `A-1042@v3#c0` (article A-1042, version 3, chunk 0) or `T-100123#c0` (ticket). Citations are compared at the document level (`A-1042@v3`), and matching is version-sensitive: citing a superseded version counts as a miss. Your final system exposes `answer(question, as_of)` in `hw5_system.py` and returns a dictionary with the keys `answer`, `citations`, `abstained` and `confidence`. The provided functions `evaluate_retrieval` and `evaluate_answers` compute the metrics below; use them unmodified.

Metric	Definition
Recall@k	Share of questions with gold evidence that have at least one gold document among the top k retrieved documents.
MRR@5	Mean of $1/\text{rank}$ of the first gold document within the top 5 (0 if absent).
Answer accuracy	Share of questions answered correctly under the type rules below.
Citation precision	Share of cited documents that are gold and current on `as_of`, over answered questions.
Stale-answer rate	Share of *stale_trap* questions answered from a superseded, draft or not-yet-effective version.
Abstention precision and recall	Abstentions measured against the *unanswerable* questions.
Latency p50 / p95	Wall-clock time per call, over at least 100 calls after a warm-up of 10.

An answer is correct when: for *answerable*, *stale_trap* and *multi_hop* questions, it contains the answer fact and does not rely on a stale version; for *unanswerable* questions, the system abstains; for *conflict* questions, where two official sources disagree, it cites both or abstains; for *injection* questions, it contains none of the injected content; for *pii_probe* questions, it contains no personal data.

General Requirements

Report measurements, not impressions. Every claim that something is better, faster or safer needs a number from your own run, with the hardware and the number of repetitions. Tune only on `golden_dev.jsonl`, and do not edit its labels. Your system must answer from its retrieved evidence: hard-coded question-to-answer or question-to-document mappings receive zero for the affected problems. The course staff may run your `hw5_system.py`, so it must build and answer without manual steps. Presenting estimated numbers as measurements is an academic-integrity violation.

Q1: Knowledge Ingestion Pipeline (30 points)

This question implements the ingestion path of the assistant: raw articles and tickets go in; a clean, versioned, privacy-safe table of retrievable chunks comes out, and it stays current as articles change. Use Spark DataFrames, as in HW1–HW3.

Problem 1: Data Contract and Profiling (8 points)

This problem implements the entry gate of the pipeline: what counts as a valid record, and which version of an article is in force.

1.1.1 Read `kb_articles.jsonl` and `tickets.csv` with explicit Spark schemas. Write validation rules for missing identifiers, empty bodies, impossible dates, invalid status or category values and duplicate `(article_id, version)` keys. Send every violating record to a quarantine table that records the rule it broke, and report the count per rule. Explain how you treat exact duplicates compared with two records that share a key but differ in content.

1.1.2 Profile the valid data: rows, versions per article, status by source, the body-length distribution, duplicate or near-duplicate bodies and the ticket category distribution. Note anything that affects later design decisions.

1.1.3 List every article with more than one active version or overlapping effective dates. State your version-resolution rule: which version is current on a given `as_of` date. The rule must cover drafts, retired versions and versions whose effective date is in the future. Keep superseded versions for auditing. Explain why two *different* official articles that disagree cannot be resolved by versioning, and how your system will treat them.

Problem 2: Cleaning, Privacy and Chunking (12 points)

This problem implements the transformation stage: clean text, remove personal data before it can be indexed, and define the chunk table that every later step reads.

1.2.1 Normalize article text: convert the HTML exported by the old content system, unescape entities, remove help-center boilerplate and normalize whitespace. Show two before-and-after examples.

1.2.2 Mask personal data in tickets before anything is chunked, embedded or indexed: email addresses, phone numbers, order numbers and customer names. Report precision and recall on `pii_sample.csv` with the provided scorer, show two cases your masker misses, and explain why masking belongs at ingestion rather than only at output.

1.2.3 Define your `knowledge_chunks` table: field names, types and purpose. It must include the chunk and document identifiers, version, category, status, source, effective date, text and a content hash. Choose a chunking strategy, run it inside Spark (for example with `mapInPandas`), and justify the chunk size with an experiment that compares two settings on Recall@3.

1.2.4 Write the chunks to Parquet partitioned by category, and report the row count of each partition.

Problem 3: Incremental Updates and Freshness (10 points)

This problem implements the update path: apply article changes as they arrive, without rebuilding everything, and measure what that costs.

1.3.1 Apply the events in `kb_updates.jsonl` one at a time, in arrival order. Your update path must be idempotent: the log delivers one event twice, and replaying it must change nothing. One event arrives later than its timestamp suggests, one is malformed, and one schedules a change for a future date. Show the event log your pipeline produces and explain how each case was handled.

1.3.2 Re-embed only chunks whose content hash has not been embedded before. Report the chunks re-processed for each event compared with a full rebuild.

1.3.3 For each development question whose note refers to an update, show the top three retrieved documents before and after the updates. Explain the result for the scheduled change (event EV-0009) on 2026-10-29 and on 2026-12-05.

1.3.4 Report per-event ingestion latency, from reading an event to its chunks being searchable (median and maximum). Then estimate the cost at Northbeam's real scale: 50,000 articles of about 1,200 tokens, 2 million tickets of about 400 tokens, 3% of articles changing each day, 400-token chunks with a 50-token overlap and 768-dimensional float32 vectors. Estimate the number of chunks, the raw vector storage and the daily re-embedding volume with and without hash-based change detection. State which stage of your pipeline would dominate at that scale and what you would change.

Q1 Submissions: In the PDF, include the quarantine counts, profile results, your version rule, the cleaning examples, the masker scores, your `knowledge_chunks` schema, the chunk-size comparison, the Parquet partition counts, the event log, the before-and-after table, and the latency and scale estimates with their arithmetic. Include readable excerpts of your validation, masking, chunking and update code.

Q2: Grounded Retrieval and Production Patterns (30 points)

This question implements the query path: find current evidence for a customer question, answer from it with citations or decline to answer, and keep the service fast and available when the model behind it misbehaves.

Problem 1: Retrieval (10 points)

This problem implements the search step that every answer depends on.

2.1.1 Run BM25 over your Q1 chunks with no filtering and report the evaluation results by question type. Classify three failures, for example as a vocabulary mismatch, a stale version, the wrong region or product line, or a distractor document.

2.1.2 Implement hybrid retrieval that combines BM25 with dense vectors, for example with reciprocal rank fusion. Any free embedding model is acceptable; state which one you use. Add a metadata filter so that only versions current on `as_of` can be returned, and decide how much to trust community posts and tickets. Customers write "Britain", "send back" and "boots" where the articles say "United Kingdom", "return" and "footwear"; consider how your retriever bridges that gap.

2.1.3 Report an ablation table with at least the rows BM25, dense, hybrid and hybrid with filtering, and the

columns Recall@1, Recall@3, MRR@5 and p50 and p95 latency. State which configuration you would ship and why.

Problem 2: Evidence-Grounded Answers (12 points)

This problem implements the answer step and its safety checks: what the assistant says, what it cites, and when it says nothing.

2.2.1 Implement `answer(question, as_of)`. An extractive answer that selects sentences from retrieved chunks is required and is sufficient for full credit. You may add an LLM, but the extractive path must work without it. Every factual sentence must cite a retrieved chunk.

2.2.2 Add a citation check. It rejects an answer that cites a chunk that was not retrieved or a version that is not current on `as_of`; the system then retries once or abstains.

2.2.3 Design an abstention policy and tune it on the development set. Plot abstention precision against recall as you vary its threshold, and mark the operating point you chose.

2.2.4 When two official articles disagree, your answer must cite both and say that they disagree, or abstain. It must not silently pick one.

2.2.5 Evaluate the *injection* and *pii_probe* questions with and without your defenses. Implement at least two mitigations, and report any answerable questions they break.

2.2.6 Report the evaluation results by question type and show five example outputs, including one each of the *stale_trap*, *conflict* and *unanswerable* types.

Three of this problem's points depend on how well your `hw5_system.py` performs when the course staff run it. It must build and answer within 15 minutes on a CPU.

Problem 3: Caching and Resilience (8 points)

This problem implements two production layers around the answer step: a cache that saves repeated work without serving stale answers, and a gateway that keeps answering when the model fails.

2.3.1 Place an exact-match cache and a semantic cache in front of `answer`. Build your knowledge base without event EV-0002, replay the first phase of `query_log.csv`, apply EV-0002, then replay the second phase. Report the hit rate of each tier, p50 and p95 latency with and without caching, and the wrong-hit rate: semantic hits whose cited documents differ from what a fresh call would cite. Compare at least two similarity thresholds, and show that no stale price is served after the update.

2.3.2 Wrap your generator in the provided `FlakyGenerator` and build a gateway with a per-call timeout, bounded retries with exponential backoff and jitter, a circuit breaker and tiered fallback (for example primary, smaller model, extractive answer, human handoff). Run 300 queries at failure rates of 0%, 10% and 30% and compare with a client that simply retries three times. Report the success rate, p50 and p95 latency and the calls served by each tier, and plot the breaker state over time for the 30% run.

Q2 Submissions: In the PDF, include the classified failures, the retrieval ablation, the answer metrics by type, the abstention plot, the five examples, the injection and PII comparison, the cache table and invalidation evidence, and the gateway table and breaker plot. Include readable excerpts of your retrieval, answer, citation-check, cache and gateway code. Move your final system into `hw5_system.py` and confirm that the check cell at the end of the notebook runs.

Q3: Mini-Project: Your Own AI Data System (40 points)

In addition to the standard tasks above, design, build and evaluate a scaled-down AI data system for a use case you choose. Write it up as a short system-design case study: requirements, scale estimate, API, high-level design, data model, the patterns you chose, testing and monitoring. Chapters 3–6 of reference [1] are worked examples of exactly this kind of write-up (an AI-native IDE, an adaptive-learning platform, e-commerce search and a customer-support agent), and Chapter 2 catalogues the patterns you can choose from. You may extend your HW4 mini-project data with a new question, or adapt one of those use cases to a different domain and dataset. The use case must differ from Northbeam: a different domain, different data and its own failure modes. Reuse your Q1 and Q2 code wherever it fits.

Problem 1: Requirements and Scale (6 points)

3.1.1 Identify the users and the decision or task the system supports. List three to five functional requirements.

3.1.2 List non-functional requirements with numeric targets, for example p95 latency, freshness, answer accuracy, abstention behaviour and cost per thousand queries. Explain where each target comes from.

3.1.3 Estimate the scale of a realistic deployment: source volume, chunks, vector storage, peak queries per second and daily ingestion. State your assumptions.

Problem 2: Design (10 points)

3.2.1 Draw a high-level design with the ingestion path, the query path, the data stores and the evaluation and monitoring loop.

3.2.2 Specify two or three API endpoints with their request and response fields.

3.2.3 Define the data model: the main tables or collections, their key fields and how versions or freshness are represented.

3.2.4 Choose at least three design patterns (for example hybrid retrieval, an ingestion pipeline with change detection, a model router, a cache, a circuit breaker, a guardrail or a privacy filter) and tie each to a requirement from Problem 1. Name one pattern you considered and rejected, and explain why.

Problem 3: Prototype (14 points)

3.3.1 Obtain the data. Document the source, license, retrieval date, size and coverage. Use at least about 200 documents or records, and include a download script where redistribution is restricted.

3.3.2 Build the ingestion pipeline with validation, cleaning, chunking and a version or freshness policy suited to your domain. Use Spark, or explain why pandas is adequate at your data size.

3.3.3 Build the query path: retrieval and grounded answers or results that cite their sources and abstain when the evidence is insufficient.

3.3.4 Implement one production pattern beyond retrieval and measure its effect.

Problem 4: Evaluation and Engineering Decision (10 points)

3.4.1 Build a labelled set of at least 30 questions or queries, including at least five unanswerable cases and the hard cases specific to your domain. Explain how you labelled them. The provided evaluation functions

work on your data if you use the same fields.

3.4.2 Compare a simple baseline with your system in a table, with one ablation and latency measurements.

3.4.3 Analyze five failures of your system, categorize them and say what you would change.

3.4.4 Write a one-page engineering decision record for your most important design choice: context, options considered, decision, evidence from your runs, consequences and what you would measure next. Mark each statement as measured or assumed. Close with a monitoring plan of five metrics, each with an alert threshold.

Mini-Project Submissions: In the PDF, include the requirements, scale estimate, design diagram, API, data model and pattern choices; the data provenance; the evaluation tables and failure analysis; and the decision record. Include your data (or a download script), code, labelled questions and generated outputs in the ZIP under `q3/`.

Grading Summary

Question	Criterion	Points
Q1	Validation rules, quarantine, profile and version-resolution rule	8
Q1	Cleaning, PII masking, chunk schema, chunk-size evidence and Parquet output	12
Q1	Idempotent updates, hash-based re-embedding, before-and-after evidence, latency and scale	10
Q2	Baseline failures, hybrid retrieval with filtering and ablation	10
Q2	Cited answers, citation check, abstention, conflicts, injection and PII handling (9); performance of `hw5_system.py` (3)	12
Q2	Cache with correct invalidation and resilient gateway	8
Q3	Mini-Project: requirements and scale estimate	6
Q3	Mini-Project: design, API, data model and justified patterns	10
Q3	Mini-Project: working prototype with data provenance and one measured production pattern	14
Q3	Mini-Project: labelled set, comparison, failure analysis and decision record	10
Total		**100**

Strong submissions measure what they claim, explain why a design was chosen over its alternatives, state what the evidence does not show, and can be reproduced from the README. A carefully measured negative result earns more credit than an unsupported claim of improvement.

Final Submission Instructions

Submit the following through CourseWorks:

1. UNI_Name_HW5.pdf — One PDF organized into Q1, Q2 and Q3 with the tables, plots, examples and code excerpts specified in each question. State the hardware used for every timing.

2. UNI_Name_HW5.zip — Your notebooks and scripts, `hw5_system.py` and any modules it imports, `requirements.txt`, your generated metric files (CSV or JSON), and a `q3/` folder with your Mini-Project data or download script, code and labelled questions. Do not include the course data files, API keys, credentials or model weights.

Write a README explaining how to reproduce every number in the PDF: the commands, the order in which to run them, library versions and the embedding model used. If any part of your system calls an external service, explain how to supply the key and confirm that `hw5_system.py` works without it. Declare any use of AI assistance as the course policy requires.

References

[1] S. Mitra, *System Design for the LLM Era: Patterns and Principles for Production-Grade AI Architecture*. Packt, 2026. Available on O'Reilly Learning: https://learning.oreilly.com/library/view/system-design-for/9781807789930/

[2] S. Robertson and H. Zaragoza, "The Probabilistic Relevance Framework: BM25 and Beyond," *Foundations and Trends in Information Retrieval*, 3(4), 2009.

[3] G. V. Cormack, C. L. A. Clarke and S. Buettcher, "Reciprocal Rank Fusion Outperforms Condorcet and Individual Rank Learning Methods," *SIGIR*, 2009.

[4] P. Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," *NeurIPS*, 2020.

[5] M. T. Nygard, *Release It! Design and Deploy Production-Ready Software*, 2nd ed. Pragmatic Bookshelf, 2018.

[6] OWASP Top 10 for Large Language Model Applications: https://genai.owasp.org/llm-top-10/

[7] Apache Spark Python API: https://spark.apache.org/docs/latest/api/python/

[8] Microsoft Presidio: https://microsoft.github.io/presidio/; sentence-transformers: https://www.sbert.net