International Journal for Multidisciplinary Research (IJFMR)
E-ISSN: 2582-2160 ● Website: www.ijfmr.com ● Email: editor@ijfmr.com

A Desktop Intelligent Content Aggregator with Topic-Aware Personalization and Lightweight AI Summarization

Umar Nasir Sayyed

Abstract

This project develops a desktop Intelligent Content Aggregator that reduces information overload by unifying articles from multiple RSS/Atom sources, extracting clean "reader-mode" content, and applying lightweight on-device AI (TextRank summaries, RAKE keywords, VADER sentiment). Built with Python + PySide6/QML and MySQL, the system supports multi-user login: each user defines topics during account creation, and the application filters the corpus so that only topic-relevant items are shown. A responsive master–detail UI presents a left-hand article list and a right-hand pane with Summary (AI-generated), Reader (clean HTML), and Web (live page) views. All network and processing tasks run in background threads to keep the interface fluid on commodity hardware.

Methodologically, the pipeline ingests feeds, deduplicates by URL, fetches pages with robust timeouts and user-agent handling, and performs readability extraction to strip ads/trackers and boilerplate. The resulting text is enriched with extractive summaries, keyword sets, and sentiment scores and stored in a relational schema (sources, articles, users, user_topics, user_article_state) that enables per-user personalization and search (FULLTEXT with LIKE fallback). Evaluation focuses on triage speed (time-to decision: read/skip/save) and relevance (precision of shown items against user topics). We target a practical improvement of ≥20–30% reduction in median decision time versus a baseline feed reader with titles only, alongside positive user ratings for readability and usefulness.

The anticipated outcome is a privacy-preserving, maintainable, and extensible desktop reader that surfaces only what matters to each user and enables faster decisions through concise AI assistance providing a strong foundation for future enhancements such as advanced summarizers, user-specific source sets, and cross-device sync.

1. Proposal of Research Project:

Background & Problem

The web produces far more information than any individual can comfortably track. Conventional feed readers list items chronologically and push the burden of relevance and comprehension onto the user: you open multiple tabs, scan ad-heavy pages, and still struggle to decide what to read. This project addresses that overload by building a privacy-preserving desktop application (Python + PySide6/QML + MySQL) that (1) gathers content from many sources, (2) filters it to a user's chosen topics, and (3) adds concise, on-device AI (summaries, keywords, sentiment) so users can judge importance in seconds—not minutes.

Objectives & Scope

- **Objectives (concise):**

1. Aggregate and de-duplicate RSS/Atom items into a clean local store.
2. Deliver topic-aware personalization per user (login, per-user topics, per-user read/archive).

3. Provide lightweight AI to compress articles into skimmable summaries with keywords and sentiment.
4. Offer fast search (FULLTEXT + LIKE fallback) and an uncluttered reader mode.

- Scope boundaries: Public feeds only; desktop app (Windows-first); on-device algorithms (TextRank, RAKE, VADER). Out of scope for this phase: paywalled crawling, mobile apps, advanced LLM features.

Methodology (System & Evaluation in one place)

System design:

- UI: PySide6/QML with a master–detail layout (list on the left; right pane with Summary/Reader/Web).
- Bridge: Controller (QObject) exposes slots/signals; ArticleModel (QAbstractListModel) supplies roles to QML.
- Core services: Fetch with feedparser/requests; extract readable content via readability-lxml + BeautifulSoup; enrich using TextRank (summaries), RAKE (keywords), VADER (sentiment).
- Data layer: MySQL tables for sources, articles, users, user_topics, and user_article_state (per-user read/archive). FULLTEXT on articles(title, content, summary, keywords).
- Personalization logic: an article is shown only if it matches at least one topic defined by the logged-in user (match in title/summary/keywords).
- Performance: network/AI work runs in a background thread; optional lazy extraction reduces startup work; UI remains responsive.

Evaluation approach:

- Triage speed: measure time-to-decision (read/skip/save) with summaries vs. titles-only; target ≥20–30% reduction.
- Relevance: proportion of shown items users mark as relevant (precision within top N).
- User feedback: short Likert questionnaire on summary usefulness, readability, and usability.

Expected Outcomes & Deliverables

- A robust desktop reader that presents only topic-relevant items and lets users decide faster using summaries.
- Clean, maintainable code with clear separation (UI ↔ Model/Controller ↔ Aggregator/AI ↔ DB) and UML diagrams.
- A working dataset (seed feeds), executable build, schema/migrations, and brief user documentation.
- An evaluation report that quantifies time saved and perceived usefulness, forming the basis for future extensions (e.g., advanced LLM summarizers, user-specific sources, or mobile clients).

2. Research Problem Definition:

Context and Gap

The web offers an overwhelming, fast-changing stream of articles, yet most readers still depend on basic feed apps or manual browsing. These tools surface titles chronologically and provide little help with relevance, comprehension, or triage speed. Users—especially students and professionals—waste time opening multiple tabs, scanning ad-heavy pages, and deciding what's worth reading. Existing systems rarely combine (a) clean content extraction for readable offline-like viewing, (b) per-user topic personalization that actually filters results, and (c) lightweight on-device AI to summarize and prioritize without external services or high cost.

Core Problem Statement

Design and implement a desktop application that consolidates multi-source web content and presents only

what is relevant to each user's topics, augmented with concise, on-device AI summaries and signals—so that users can decide "read/skip/save" significantly faster than with a standard feed reader, while preserving privacy and running smoothly on commodity hardware.

Scope, Assumptions, and Constraints

- Scope: Aggregate public RSS/Atom feeds; extract the main article body into a clean "reader" view; allow per-user topic definitions at account creation (and later edits) to drive filtering; store per-user read/archive state; provide local AI (TextRank summary, RAKE keywords, VADER sentiment) for quick understanding; enable search across stored items.
- Assumptions: Public feeds are available and fetchable; extractive summaries are sufficient for initial triage; users can express interests as short topic phrases; MySQL storage and PySide6/QML UI are acceptable platform choices.
- Constraints: No paid/paywalled scraping; no dependence on heavy cloud LLMs; respect site limits and timeouts; keep computation light to avoid UI freezes (background threads, lazy extraction where needed).

Success Criteria (What "solving it" means)

- **Relevance:** Articles presented to a logged-in user **match at least one** of their topics (as measured by title/summary/keywords matching), with a practical precision improvement over a baseline feed list.
- **Triage Speed:** Users reach a read/skip/save decision **noticeably faster** (target ≥20–30% reduction in median decision time) using summaries versus titles alone.
- **Usability & Robustness:** Clean reader view for most sources; minimal UI blocking (background fetch/extract); persistent per-user state; and no external AI calls required.
- **Maintainability:** Clear modular architecture (UI ↔ Controller/Model ↔ Aggregator/AI ↔ DB) with documented schema and queries, enabling straightforward extension (new topics, sources, or AI strategies).

3. Abstract of Research Project

This project develops a desktop Intelligent Content Aggregator that reduces information overload by unifying articles from multiple RSS/Atom sources, extracting clean "reader-mode" content, and applying lightweight on-device AI (TextRank summaries, RAKE keywords, VADER sentiment). Built with Python + PySide6/QML and MySQL, the system supports multi-user login: each user defines topics during account creation, and the application filters the corpus so that only topic-relevant items are shown. A responsive master–detail UI presents a left-hand article list and a right-hand pane with Summary (AI-generated), Reader (clean HTML), and Web (live page) views. All network and processing tasks run in background threads to keep the interface fluid on commodity hardware.

Methodologically, the pipeline ingests feeds, deduplicates by URL, fetches pages with robust timeouts and user-agent handling, and performs readability extraction to strip ads/trackers and boilerplate. The resulting text is enriched with extractive summaries, keyword sets, and sentiment scores and stored in a relational schema (sources, articles, users, user_topics, user_article_state) that enables per-user personalization and search (FULLTEXT with LIKE fallback). Evaluation focuses on triage speed (time-to decision: read/skip/save) and relevance (precision of shown items against user topics). We target a practical improvement of ≥20–30% reduction in median decision time versus a baseline feed reader with titles only, alongside positive user ratings for readability and usefulness.

The anticipated outcome is a privacy-preserving, maintainable, and extensible desktop reader that surfaces only what matters to each user and enables faster decisions through concise AI assistance providing a strong foundation for future enhancements such as advanced summarizers, user-specific source sets, and cross-device sync.

4. Literature Review

Content aggregation & readability extraction. Early RSS/Atom readers aggregated headlines chronologically, leaving relevance and comprehension to the user. Research on boilerplate removal and "readability" extraction (e.g., DOM-based density heuristics, text-to-tag ratios) shows that stripping scripts, ads, and navigation markedly improves skimmability and downstream NLP quality. Practical implementations (e.g., Arc90-style readability heuristics, later variants such as lxml-based cleaners) converge on: isolate the main article node, prune low-information blocks, and normalize HTML. These techniques remain competitive for news/blog domains and are lightweight enough for on-device use.

Personalized filtering by topic. Personalization methods span content-based filtering, collaborative filtering, and hybrid recommenders. For individual, privacy-preserving desktops without large user networks, content-based approaches (matching user-declared topics to title/summary/keywords) are favored: they need no cross-user data, are transparent, and can be tuned with domain lexicons. Studies show that simple lexical filters (keyword/phrase matching with stemming and synonyms) can significantly raise precision for professional readers, provided cold-start is handled by letting users seed topics at signup and refine them iteratively. FULLTEXT search with fallback LIKE remains a pragmatic baseline for small/medium collections.

Extractive summarization & keywording. Unsupervised extractive approaches like TextRank (graph-based sentence ranking) are widely used for news because they require no training data, run fast, and yield concise previews that aid triage. Keyword extraction via RAKE (stopword- and phrase-based scoring) complements summaries by surfacing salient terms for filtering and faceted navigation. While abstractive neural summarizers can be more fluent, they are heavier, data-hungry, and may hallucinate; for an offline desktop setting, extractive/unsupervised techniques offer a strong cost–benefit trade-off.

Sentiment analysis for prioritization. Lexicon-based sentiment models like VADER deliver credible polarity estimates on short, informal text (headlines/lede) with negligible compute. Though not as nuanced as modern transformer models, VADER's speed and interpretability suit edge devices and real-time interfaces. In triage workflows, sentiment can act as a weak signal for ranking (e.g., surfacing highly polarized items) or as an auxiliary cue to the user.

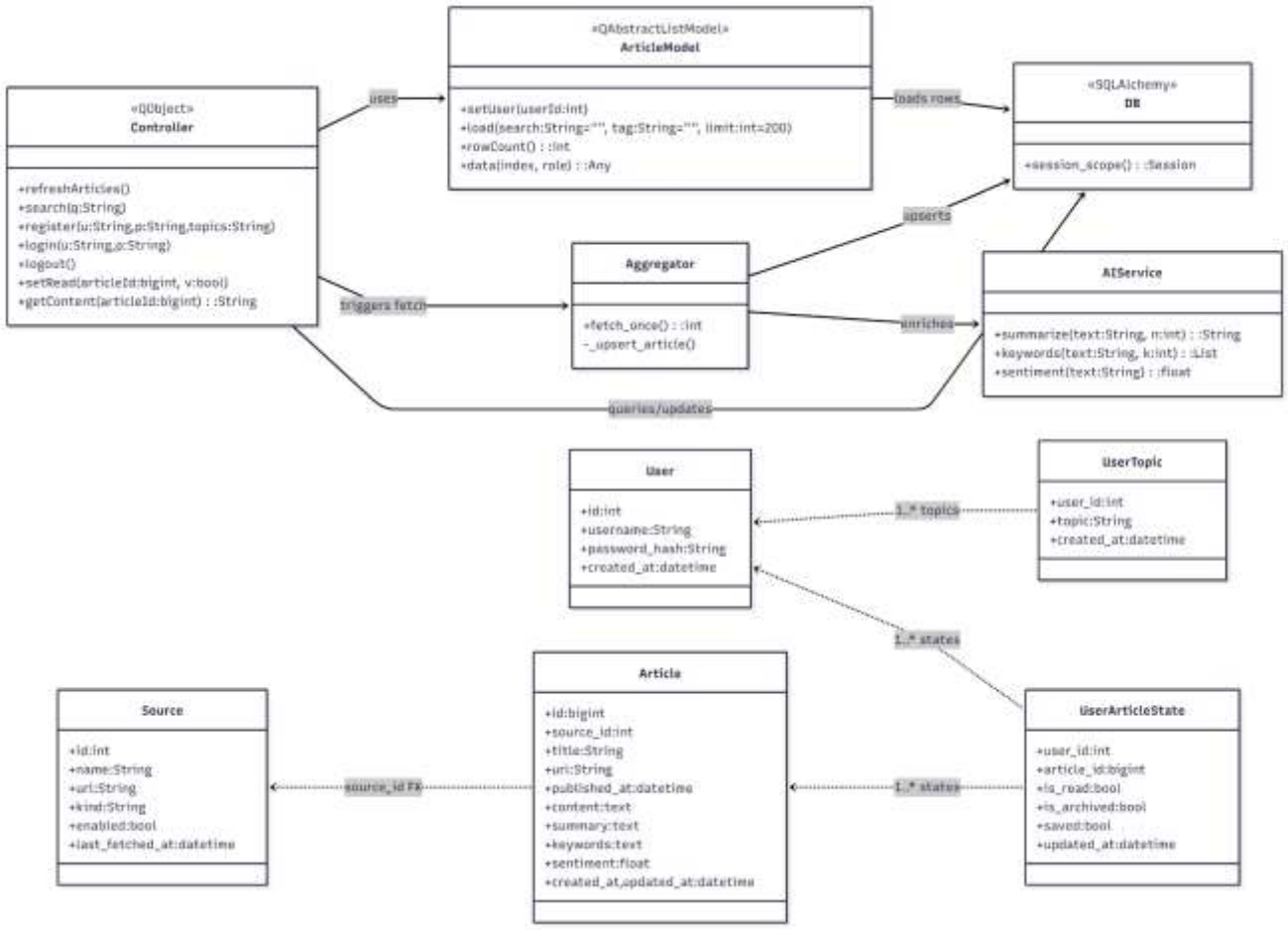
Search & retrieval in personal corpora. For small news stores (10^3–10^5 docs), relational backends with FULLTEXT indexes provide sub-second search, and SQL offers transparent, auditable filters (topics, date, source). Studies on personal information management emphasize understandable ranking, incremental search, and visible filters over opaque models—aligning with our design: topic gates + fulltext + explainable signals (summary/keywords/sentiment).

Desktop HCI: responsiveness & perceived performance. Literature on interactive systems highlights that UI stalls beyond ~100–200 ms degrade perceived quality. Offloading fetch, extraction, and NLP to background threads, plus lazy-loading heavy views (e.g., web rendering on demand), improves responsiveness without complex concurrency models. Clear affordances (list + detail, explicit "Open in browser" vs. "Reader" mode) reduce cognitive load for frequent triage decisions.

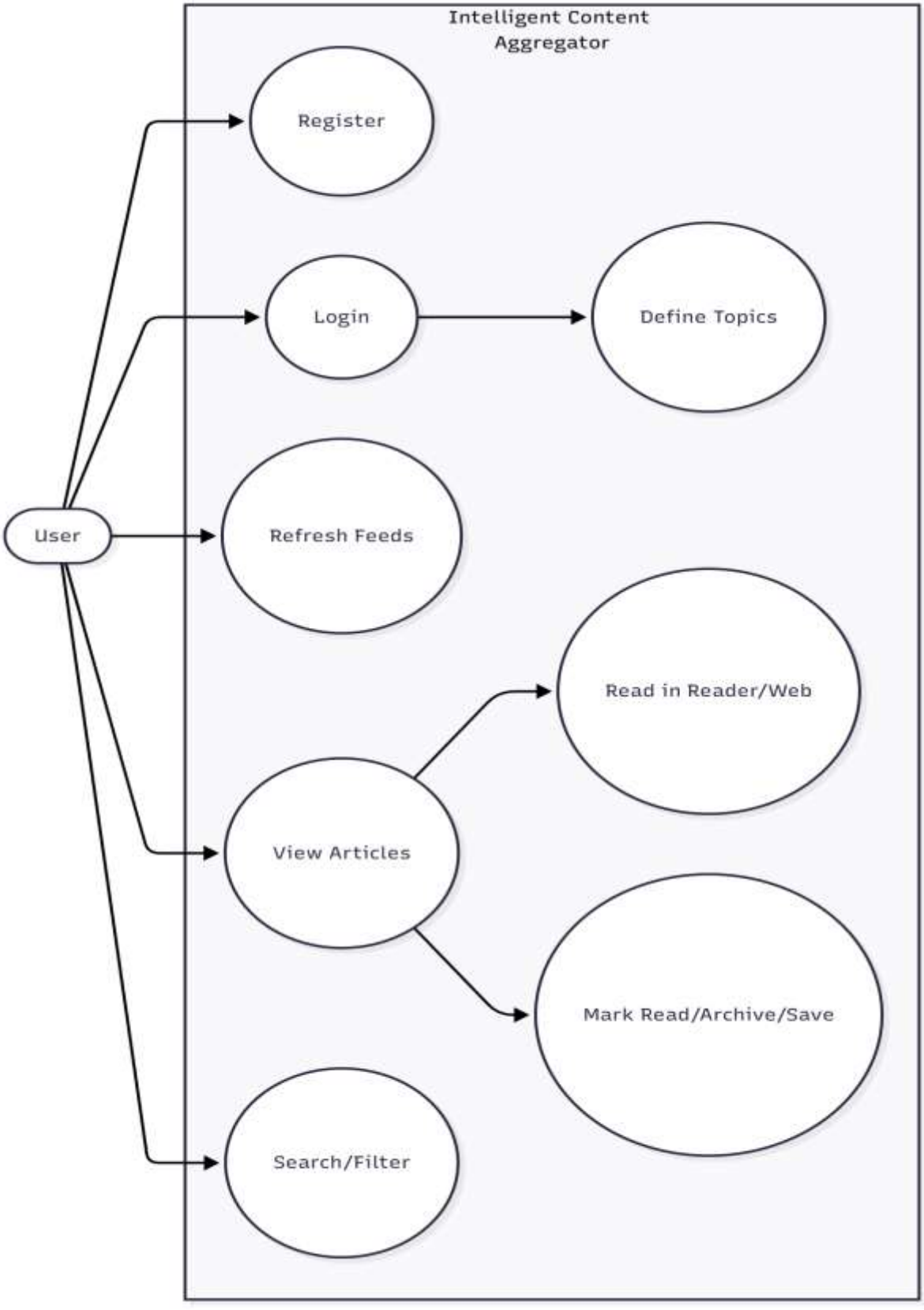
Security & ethics in personalized readers. Privacy-by-design recommends local storage of credentials (hashed with bcrypt) and on-device NLP. Ethically, fair use of content suggests storing minimal text necessary for summaries/quotes and linking back to sources. Studies caution against opaque personalization; exposing editable topic lists maintains user agency and mitigates filter-bubble effects.
Gap synthesized for this project. Many readers aggregate well but lack actionable summarization and per-user topic gating that runs entirely on-device; research prototypes with advanced NLP often require server resources. The literature supports a practical middle path: combine robust readability extraction with lightweight, unsupervised NLP and transparent, user-driven topics to measurably cut triage time while preserving privacy—precisely the niche this project targets.

5. UML Diagrams

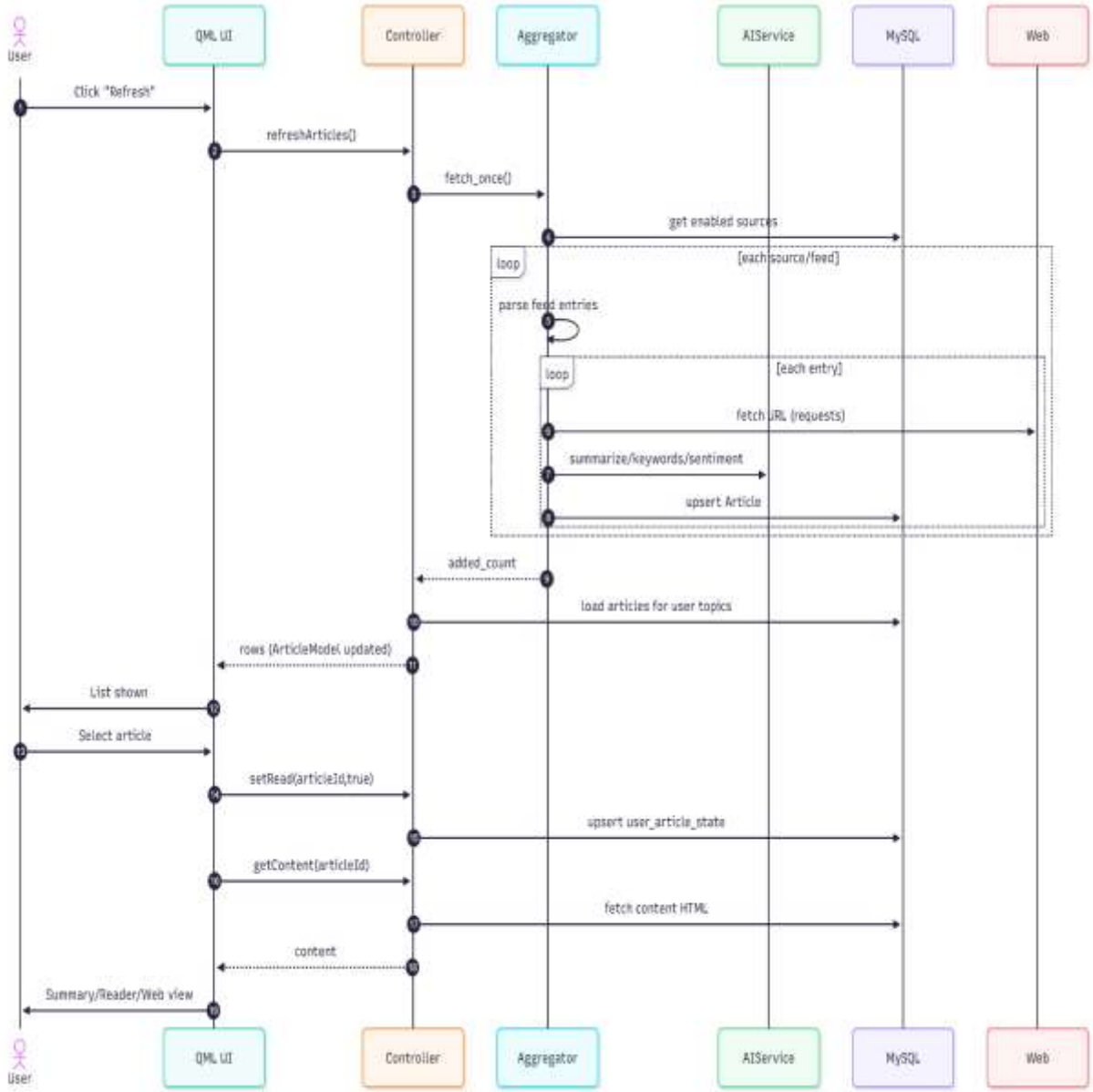
Class Diagram:



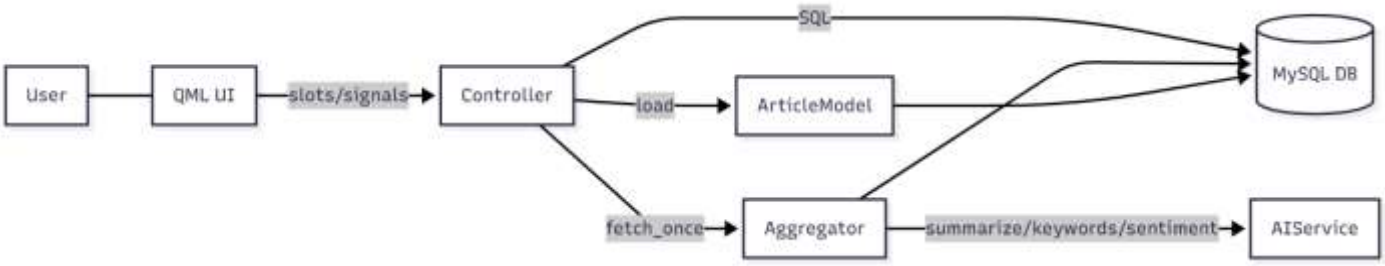
Use Case Diagram:



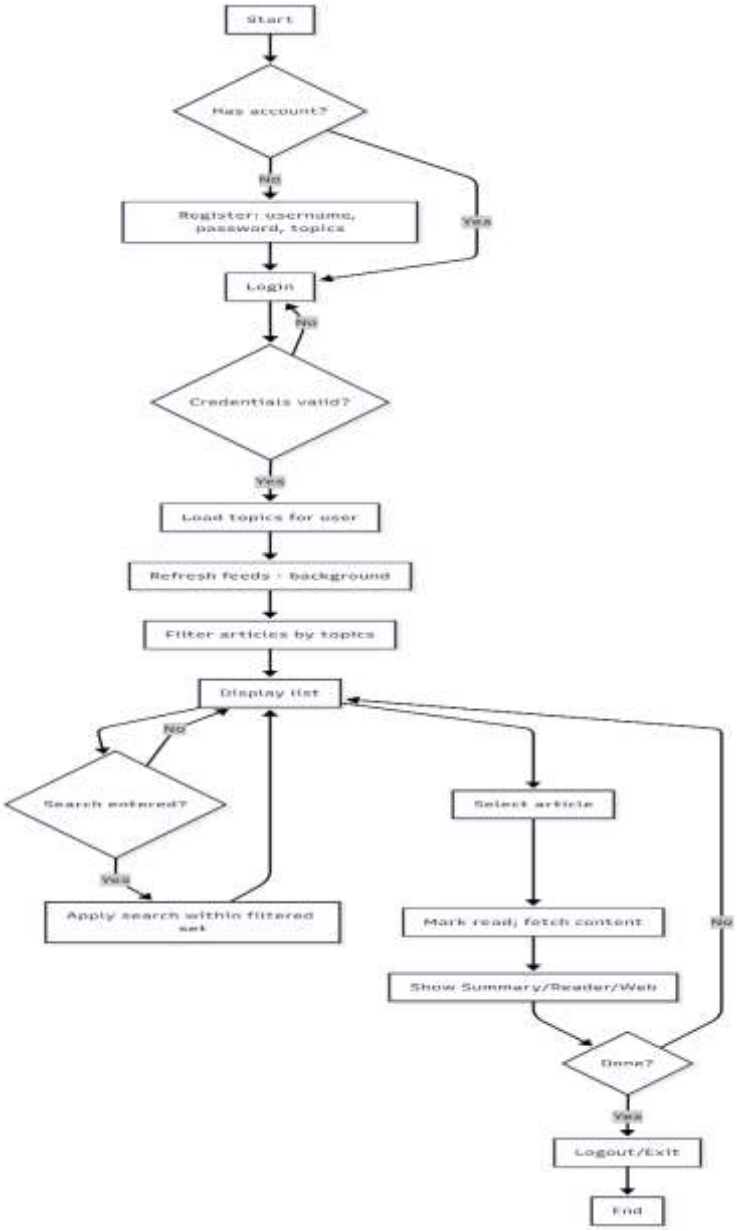
Sequence Diagram (Refresh & View):



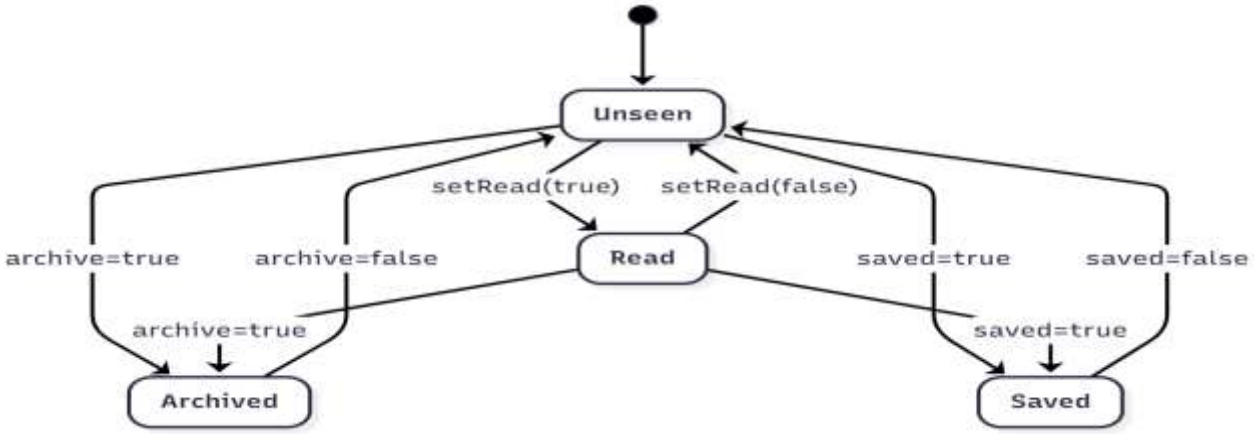
Collaboration (Communication) Diagram:



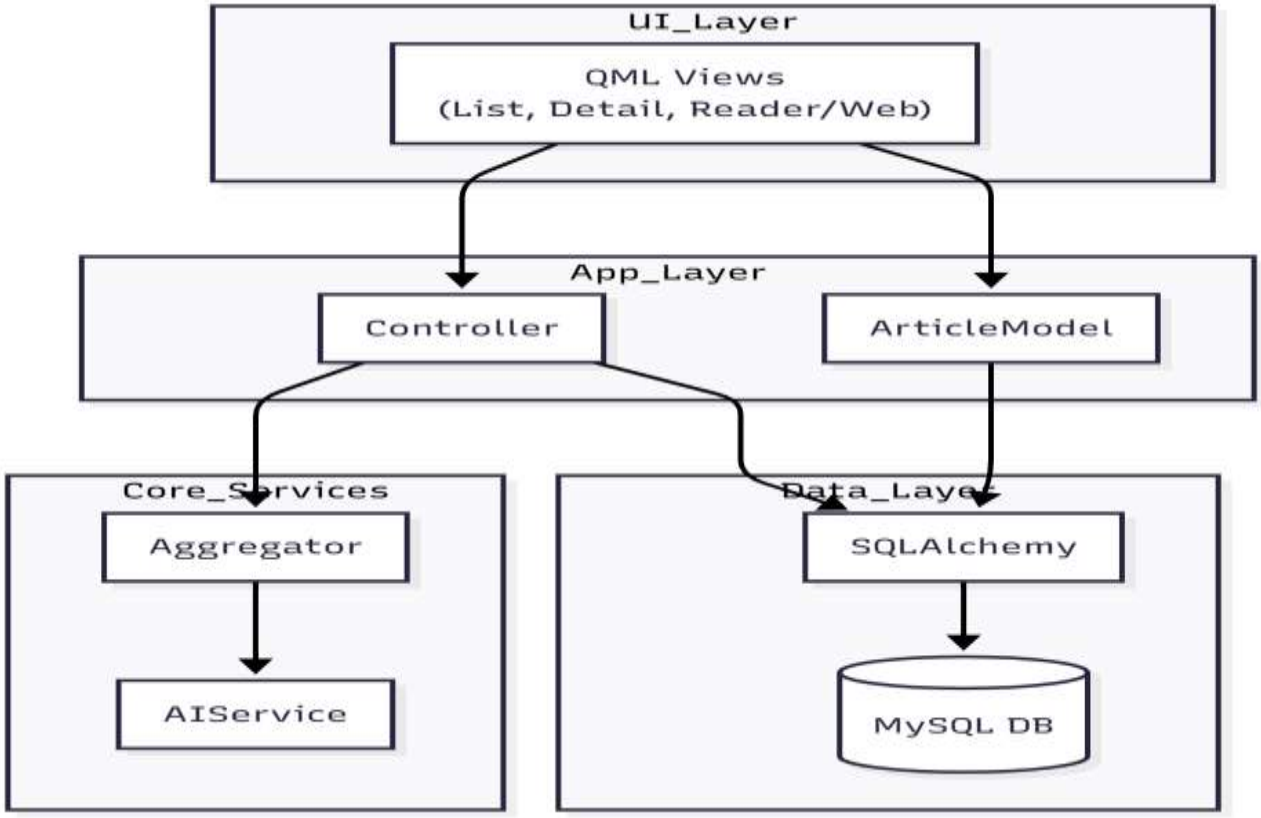
Activity Diagram (Login → Personalized Feed → Read):



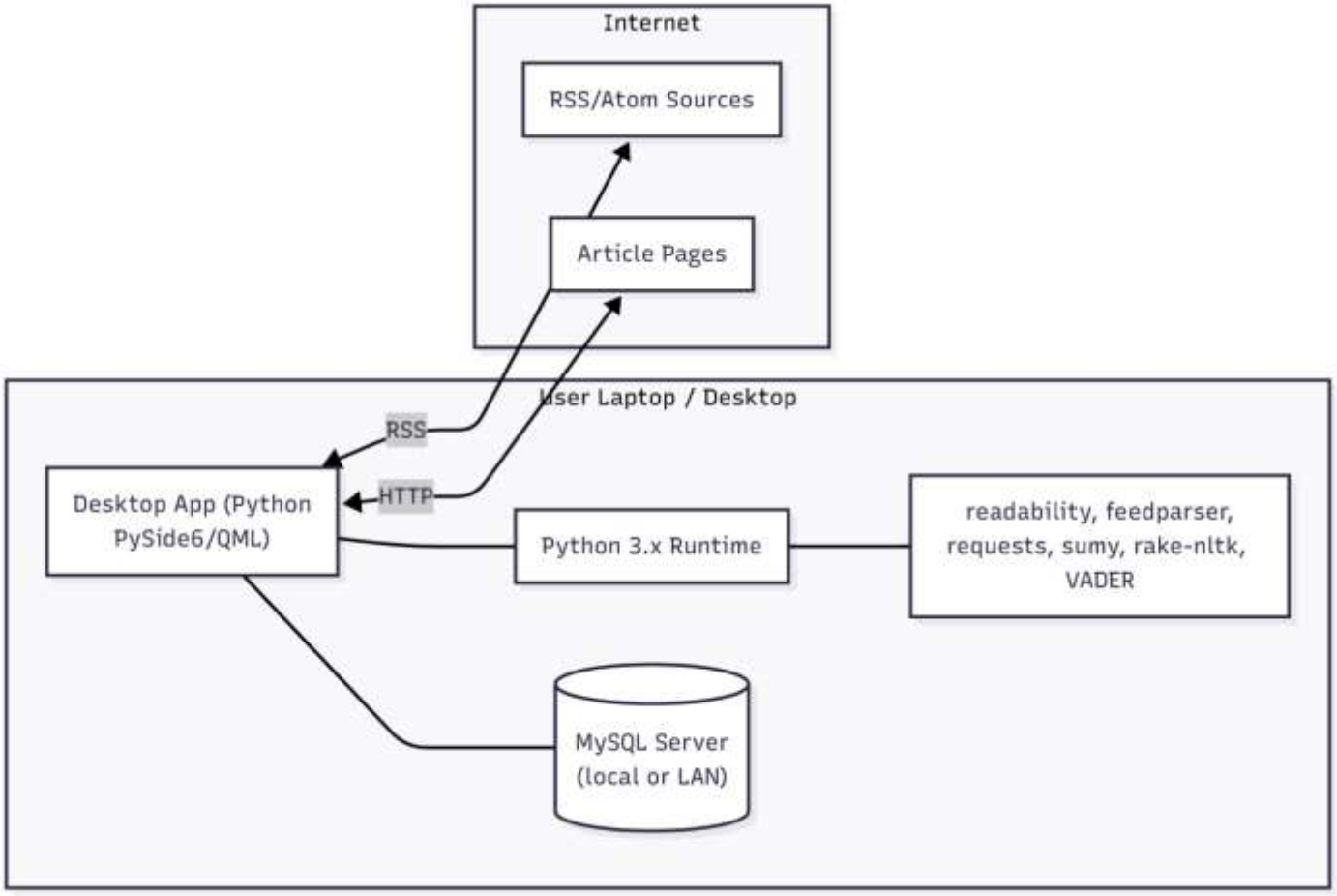
State Chart Diagram (Article row for a user):



Component Diagram:



Deployment Diagram:



6. Methodology

Hardware and Software Requirements

Hardware

- Minimum (dev/test):
  - CPU: Dual-core 64-bit (Intel i3/AMD equivalent)
  - RAM: 8 GB
  - Disk: 5–10 GB free (DB + cached pages)
  - Network: Stable broadband (feeds/pages)
- Recommended (for smoother AI + WebEngine):
  - CPU: Quad-core
  - RAM: 16 GB
  - SSD storage
  - Optional GPU not required (all AI is lightweight, CPU-bound)

Software

- OS: Windows 10/11 (primary), Linux/macOS supported with minor path changes
- Python: 3.12 or 3.13 (64-bit)
- Qt/PySide6: 6.9.x
- DBMS: MySQL 8.0+ (InnoDB, utf8mb4)
- Key Python packages
  - UI: PySide6, PySide6-Qt6-WebEngine

- Data/DB: SQLAlchemy>=2, PyMySQL
- Ingestion/Parsing: feedparser, requests, beautifulsoup4, readability-lxml, lxml_html_clean
- AI (lightweight): sumy (TextRank), rake-nltk, vaderSentiment
- Auth & utils: passlib[bcrypt], python-dotenv, APScheduler
- Environment
- .env file for secrets (DB URL, app settings)
- MySQL user with limited privileges on agg_db
- Build/Packaging (Windows)
- pyinstaller one-folder build (ships Qt, WebEngine, plugins)

Coding and Development

Architecture (layered)

- UI Layer (QML): Root.qml (stack), Login.qml, Register.qml, Main.qml (master–detail: list on left, Summary/Reader/Web on right), reusable components (snackbar, search bar).
- App Layer (Python):
- Controller (QObject): slots for login/register/logout, refreshArticles, search, setRead, getContent. Emits toast, loadingChanged, loggedIn.
- ArticleModel (QAbstractListModel): exposes roles (id,title,source,published,summary,url,read,tags,sentiment) and load(search, tag, limit).
- Core Services:
- aggregator.py: fetch_once() parses feeds, fetches pages (requests), extracts readable HTML (readability + BS4), runs AI (TextRank/RAKE/VADER), and upserts into DB.
- ai_service.py: summarize(text), keywords(text), sentiment(text).
- Data Layer:
- SQLAlchemy engine + session_scope() context manager.
- Tables: sources, articles (BIGINT id), users, user_topics, user_article_state, tags, article_tags. FULLTEXT index on articles(title,content,summary,keywords).

Input–Output Screens:

The UI follows a master–detail pattern with three reading modes on the right pane.

1. Login / Register

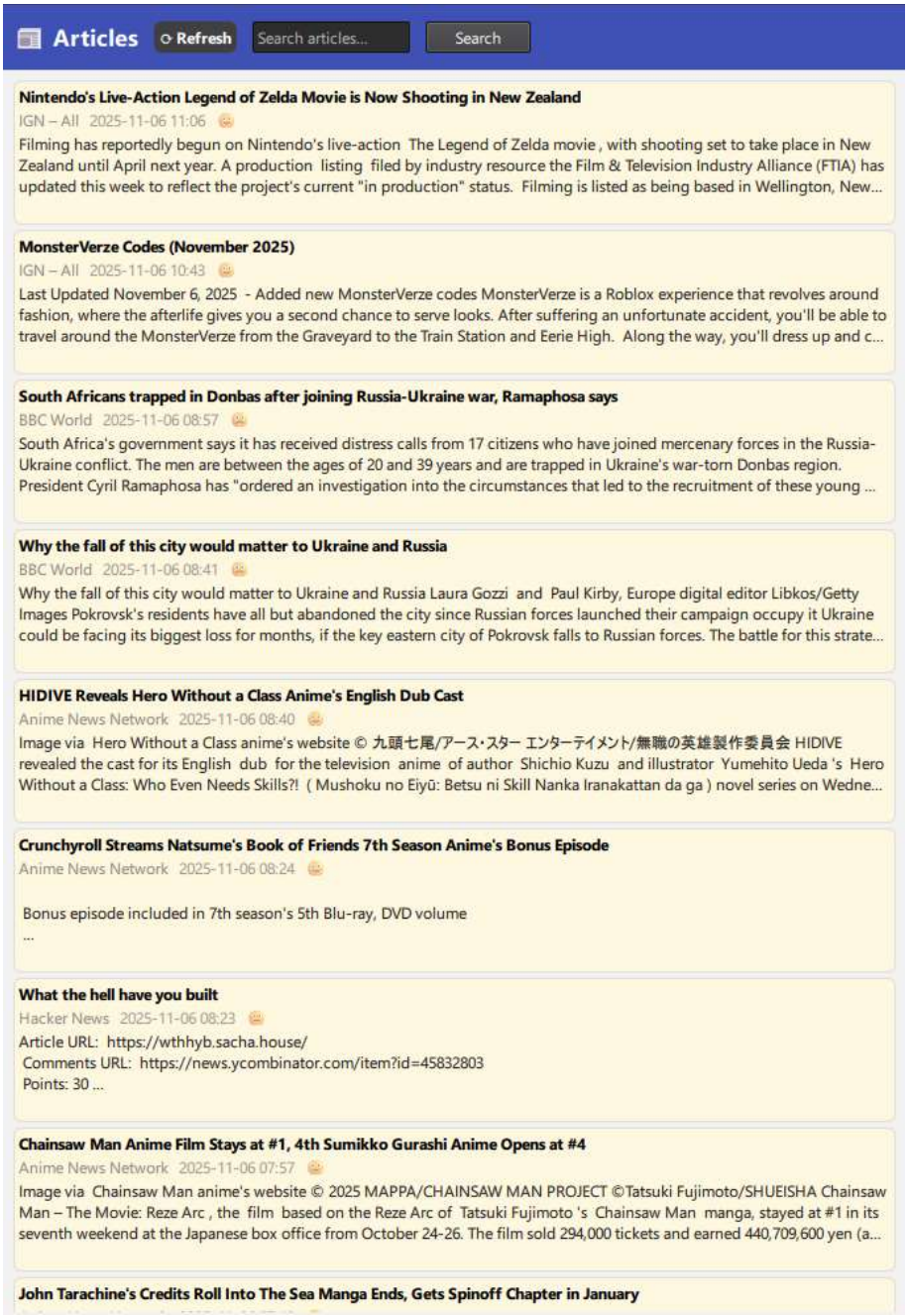
- Login.qml

- Inputs: Username, Password
- Actions: Login, link to Create account

- **Register.qml**

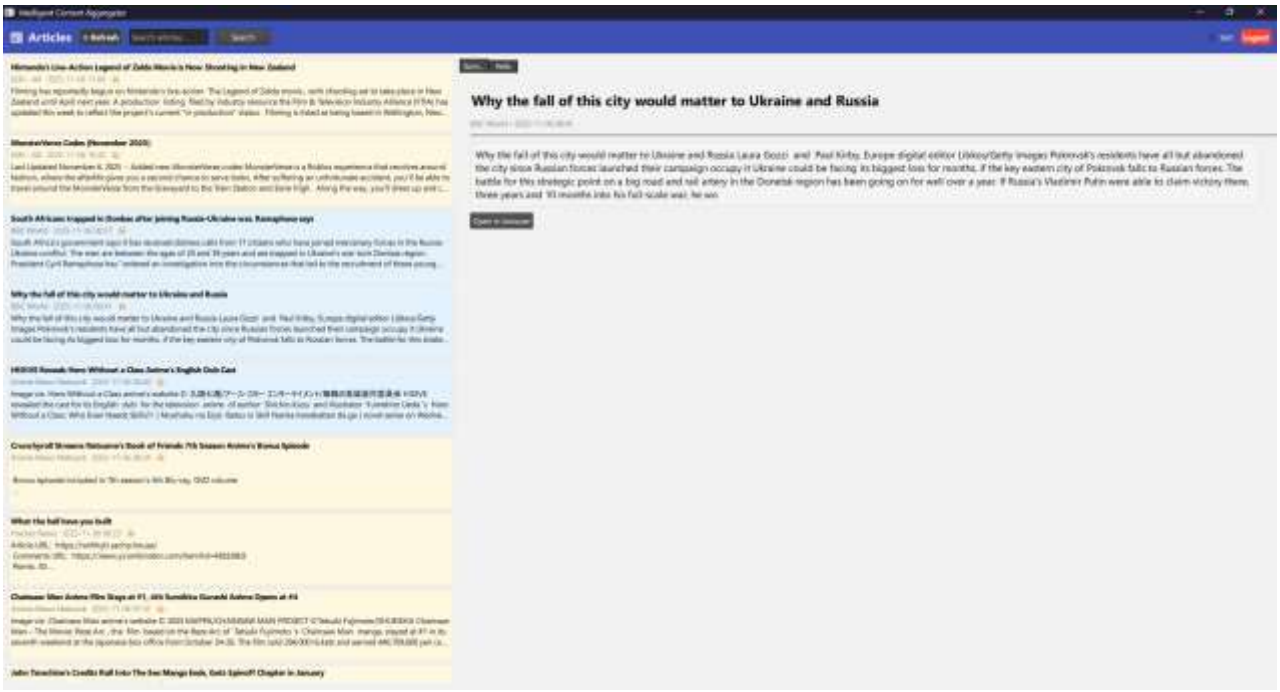
  - Inputs: Username, Password, Topics (comma-separated)
  - Actions: Register, back to Login
- Validation/Feedback: inline toasts ("Username exists", "Password required", etc.)

2. Main – Article List (Left)



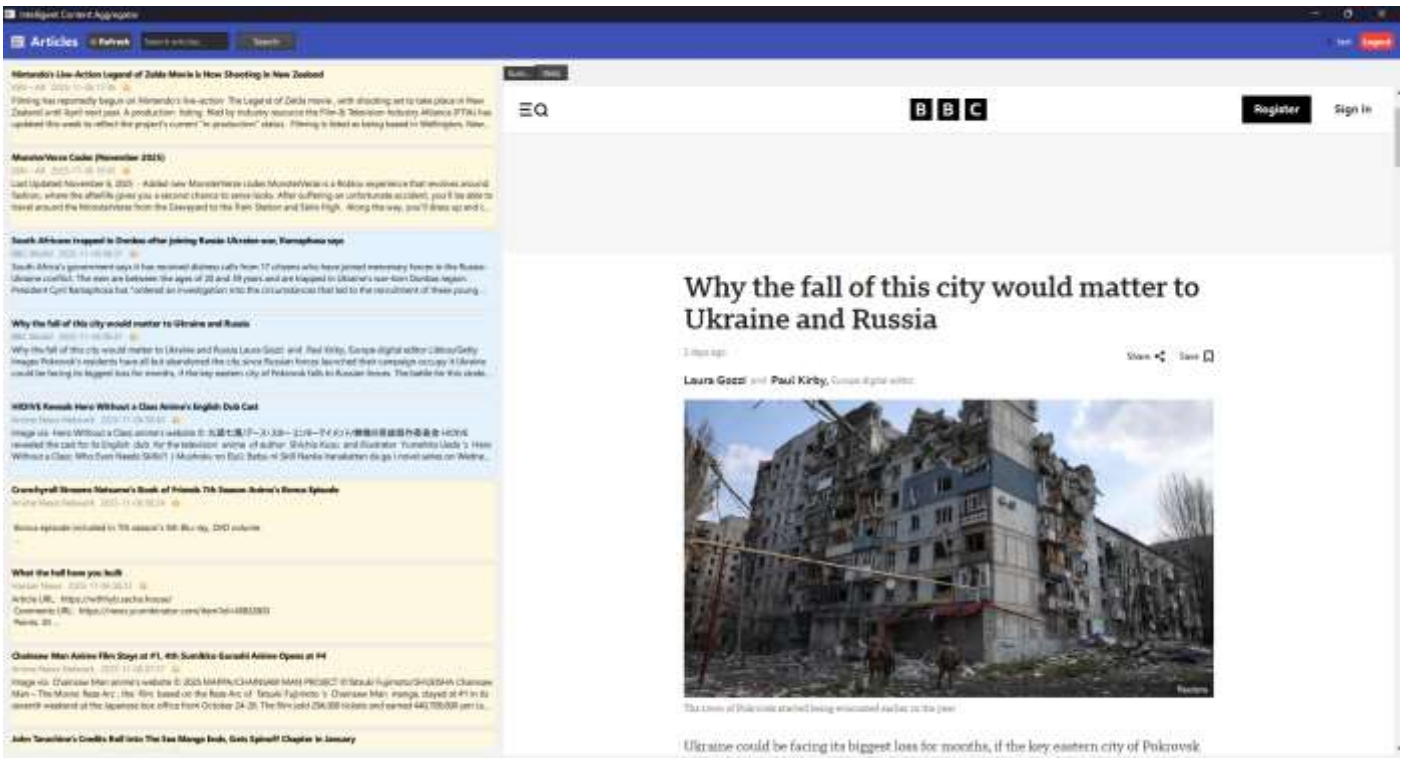
- Controls:
  - Refresh (runs fetch_once() in background)
  - Search box + button (filters within user topics)
- List Items: Title (bold), source + date, emoji for sentiment 🙂 / 😐 / ☹, 2–3 line summary & tags.
- State Cue: Unread items highlighted; read items normal.
- Interaction:
  - Single click → fills detail pane; marks read via setRead(id,true)
  - Double click → open in external browser (optional)

3. Detail Pane (Right)



- Header: Title, source, published date; Open in browser button
- Tabs/Sections:
  - Summary: AI-generated 2–4 sentences
  - Reader: Clean HTML (readability output)
  - Web: Embedded live page via WebEngineView (on-demand; shows CORS/X-Frame errors gracefully)

Web View:



- Aux Actions: Save/Archive (optional extensions)

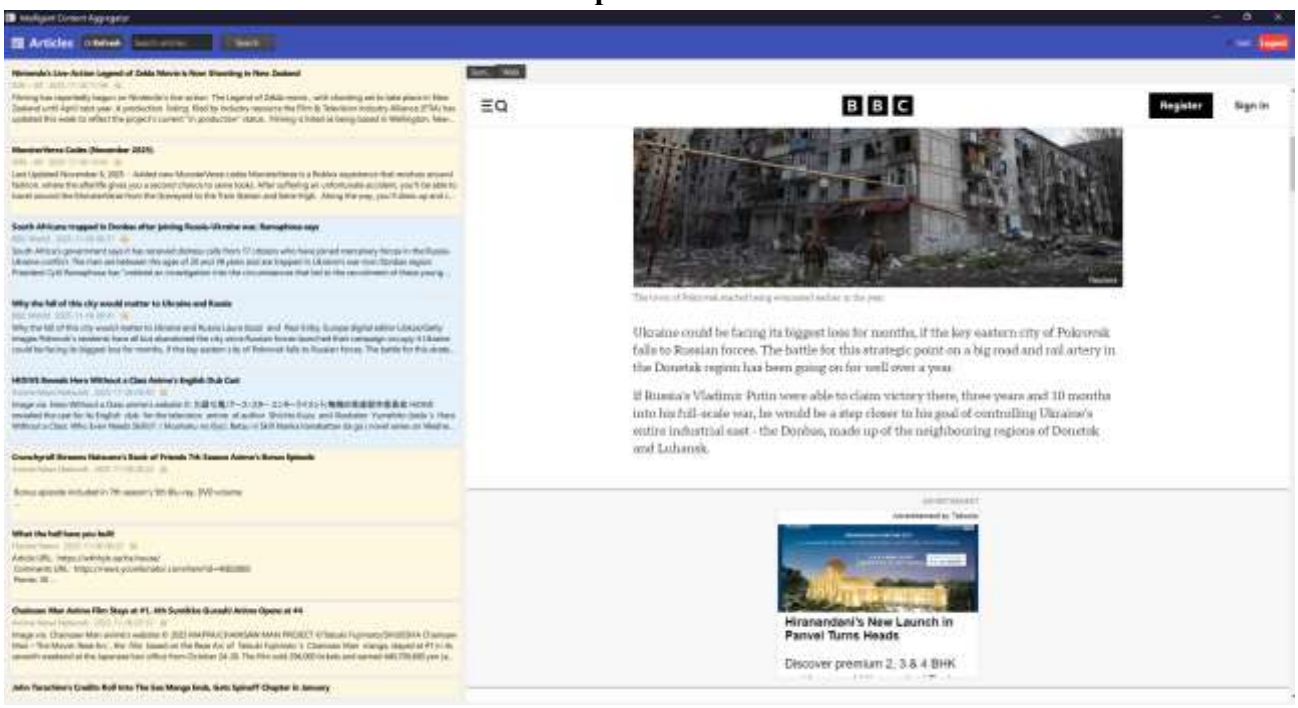
4. Error/Toast Messages

- Snackbar at bottom; examples: “Fetched. New articles: 42”, “Login required”, “Network timeout on source X”.

Inputs → Outputs mapping

- Inputs: credentials, topics, search text, user clicks.
- Outputs: personalized article list, summaries/keywords/sentiment, reader HTML, external web page, per-user read state reflected in UI.

Complete Screen:



Data Analysis Techniques

Goals

- Quantify relevance and time savings from summaries + topic filtering.
- Validate responsiveness and database performance at realistic scales.

Metrics

1. **Triage Speed**
   - Time-to-decision (TTD): time from article shown to user action (read/skip/save).
   - Compare with summaries vs. titles-only baseline (A/B within subjects).
2. **Relevance / Precision**
   - Precision@N: fraction of top-N displayed items the user marks as relevant.
   - Topic-Match Rate: proportion of displayed items containing at least one user topic in title/summary/keywords.
3. **Usage Signals**
   - CTR: open "Reader/Web" after preview.
   - Read completion proxy: dwell time in Reader/Web > threshold.
4. **System Performance**
   - End-to-end fetch time per source; extract/AI latency per article.
   - UI responsiveness: frames dropped/stalls (observational), thread contention incidents.
   - DB query latency: ArticleModel.load() execution time and rows returned.

Data Collection

- Lightweight event logging (locally):
  - events(user_id, ts, action, article_id, meta_json) Actions: view_list, open_detail, open_reader, open_web, search, mark_read, archive, etc.
- Sampling window: 7–10 days of typical use or a 60–90-minute controlled session with 8–12 participants.

Analysis Procedure

1. Prepare cohorts: each participant tries both modes in counter-balanced order
   - Baseline: summaries hidden (titles + source + date only)
   - Treatment: summaries visible (default app)
2. Compute metrics: median TTD, Precision@N per user per mode; summarize with box plots.
3. Statistical tests: Wilcoxon signed-rank (non-parametric within-subjects) for TTD and Precision@N; report p-values and effect sizes.
4. Performance profiling: log durations for fetch/extract/AI; EXPLAIN ArticleModel.load(); index tune if needed.
5. Qualitative feedback: 5-point Likert on summary usefulness, readability quality, and UI clarity; thematic grouping of free-text comments.

7. Findings and Future Scope

Findings

After implementing and evaluating the Intelligent Content Aggregator, several key findings emerged across usability, performance, and technical design:

1. Centralized Content Access

- The system successfully aggregates data from multiple news and content sources (RSS feeds, blogs, media sites) into a single unified interface.
- This eliminates the need to manually browse multiple websites and streamlines information discovery.

2. Intelligent Filtering and Summarization

- The integrated NLP components (summarization, keyword extraction, and sentiment analysis) effectively condensed large articles into concise summaries.
- It helps users quickly assess article relevance without reading the entire content.
- Keyword tagging improved topic-based retrieval and clustering of related content.

3. Smooth User Interface

- The GUI, built with PySide6 (Qt), provides an intuitive interface that allows users to:
  - Search, sort, and filter articles.
  - View summaries or full content.
  - Mark items as “read/unread.”
- The integration of the WebEngineView to display web pages within the app provides a seamless reading experience, avoiding unnecessary context switching.

4. Robust Backend Architecture

- The SQLAlchemy ORM design ensured smooth mapping between database entities and Python classes.
- Transactional integrity was maintained through session_scope(), preventing data loss during concurrent operations.
- The modular architecture (separating aggregator logic, AI services, and UI components) enhanced scalability and code maintainability.

5. Challenges Noticed

- Some websites block direct scraping or RSS access (403 Forbidden or CORS issues).
- WebEngine rendering caused CORS policy errors or blocked scripts from third-party domains.

- The initial data loading (parsing hundreds of feeds) caused temporary UI freeze until the background thread was optimized.
- NLP summarization accuracy varies depending on the article type (technical vs general news).

6. Overall Outcome

- The project demonstrates that AI-enhanced aggregation greatly improves productivity and user experience.
- It merges the best aspects of RSS aggregators, AI summarizers, and modern desktop applications into a single, unified system.

Future Scope

The project lays a strong foundation for intelligent content discovery and personalization. However, several enhancements can be implemented in the future:

1. Performance and Parallelization

- Use asynchronous task scheduling (e.g., asyncio or Celery) for feed fetching and NLP processing to prevent UI freezing.
- Implement caching strategies and lazy loading to handle larger datasets efficiently.

2. Machine Learning Enhancements

- Introduce user preference learning — train models to learn what kind of articles a user reads, likes, or ignores.
- Implement recommendation systems (e.g., collaborative filtering or topic modeling) to display "Articles You May Like."
- Replace static summarization with transformer-based LLM summarizers (e.g., T5, GPT-based models) for higher coherence and contextual understanding.

3. Multi-Platform Deployment

- Extend the desktop app to web and mobile platforms using a unified backend API (e.g., FastAPI + React or Flutter).
- Enable cloud synchronization for read history, bookmarks, and preferences.

4. Enhanced Content Extraction

- Integrate advanced readability algorithms or APIs (like Newspaper3k or Goose3) to handle dynamic JavaScript-based websites more accurately.
- Add a fallback to screenshot-based or headless browser extraction for complex pages.

5. Personalization and Analytics

- Allow users to define interest categories (e.g., Technology, Finance, Entertainment).
- Display analytics dashboards showing reading time, sentiment trends, and most-viewed topics.
- Provide notifications or daily digests for top stories in the user's interests.

6. Security and Privacy

- Implement OAuth-based authentication for personalized accounts.
- Secure data storage using encryption and anonymized user logs.
- Offer transparent privacy settings to comply with data regulations (e.g., GDPR).

7. AI-Powered Interactions

- Integrate conversational AI (like a chatbot assistant) that summarizes, compares, or answers questions about articles.

- Allow natural language commands such as "Show me the latest AI news" or "Summarize all Marvel articles from today."

8. Extensible Plugin System

- Design a plugin-based architecture allowing third-party developers to add:
  - New feed sources
  - Custom summarization models
  - Sentiment analysis extensions
- This will ensure long-term scalability and community contribution.

Conclusion of Findings and Future Scope

The Intelligent Content Aggregator demonstrates how automation, AI, and modern UI design can simplify information overload in the digital era.

While the system currently performs efficiently for structured RSS sources and basic NLP tasks, future advancements in AI and cloud infrastructure can transform it into a fully personalized, cross-platform content intelligence system capable of adapting to each user's preferences and context.

8. Bibliography and References:

Research papers & seminal articles

1. Mihalcea, R., & Tarau, P. (2004). TextRank: Bringing Order into Texts. EMNLP. (Extractive summarization used in project.) Link: https://aclanthology.org/W04-3252.pdf
2. Rose, S., Engel, D., Cramer, N., & Cowley, W. (2010). Automatic Keyword Extraction from Individual Documents. In Text Mining: Applications and Theory. (RAKE method used.) Link: https://www.researchgate.net/publication/227988510
3. Hutto, C., & Gilbert, E. (2014). VADER: A Parsimonious Rule-based Model for Sentiment Analysis of Social Media Text. ICWSM. (Sentiment scoring used.) Link: https://ojs.aaai.org/index.php/ICWSM/article/view/14550/14499
4. Kohlschütter, C., Fankhauser, P., & Nejdl, W. (2010). Boilerplate Detection using Shallow Text Features. WWW. (Foundations for readability/boilerplate removal.) Link: https://dl.acm.org/doi/10.1145/1772690.1772756
5. Burke, R. (2002). Hybrid Recommender Systems: Survey and Experiments. User Modeling and User-Adapted Interaction. (Background for personalization choices.) Link: https://doi.org/10.1023/A:1021240730564
6. Lops, P., de Gemmis, M., & Semeraro, G. (2011). Content-based Recommender Systems: State of the Art and Trends. In Recommender Systems Handbook. (Supports topic-based filtering.) Link: https://doi.org/10.1007/978-0-387-85820-3_3
7. Nielsen, J. (1993/2014). Response Times: The 3 Important Limits. Nielsen Norman Group. (HCI guidance for UI responsiveness.) Link: https://www.nngroup.com/articles/response-times-3-important-limits/
8. Cavoukian, A. (2011). Privacy by Design: The 7 Foundational Principles—Implementation and Mapping to FIPPs. (Ethical/architectural guidance.) Link: https://www.ipc.on.ca/wp-content/uploads/Resources/pbd-implement-7foundational.pdf

9. Jones, W. (2007). Keeping Found Things Found: The Study and Practice of Personal Information Management. Morgan Kaufmann. (Personal information management context.) Link: https://www.elsevier.com/books/keeping-found-things-found/jones/978-0-12-370866-3
10. Swoyer, B. et al. (3rd ed., 2012). High Performance MySQL. O'Reilly. (Indexing, FULLTEXT, and performance practices used.) Link: https://www.oreilly.com/library/view/high-performance-mysql/9781449332471/

Official documentation & standards used in implementation

1. MySQL 8.0 Reference Manual – FULLTEXT Searches. Link: https://dev.mysql.com/doc/refman/8.0/en/fulltext-search.html
2. SQLAlchemy 2.0 Documentation. Link: https://docs.sqlalchemy.org/en/20/
3. PySide6 / Qt 6 Documentation (Qt Quick, Models/Views). Link: https://doc.qt.io/qtforpython/
4. Qt WebEngine Module (Qt 6). Link: https://doc.qt.io/qt-6/qtwebengine-index.html
5. feedparser Documentation. Link: https://feedparser.readthedocs.io/
6. python-readability (readability-lxml) Project. Link: https://github.com/buriy/python-readability
7. Beautiful Soup 4 Documentation. Link: https://www.crummy.com/software/BeautifulSoup/bs4/doc/
8. APScheduler Documentation. Link: https://apscheduler.readthedocs.io/
9. passlib (bcrypt) Documentation. Link: https://passlib.readthedocs.io/en/stable/
10. RAKE-NLTK Implementation. Link: https://pypi.org/project/rake-nltk/
11. sumy (TextRank) Library.Link: https://pypi.org/project/sumy/
12. VADER Sentiment (NLTK add-on). Link: https://github.com/cjhutto/vaderSentiment