End-to-End QA Generation for Thrive@Pitt: Robust Data Preparation, Quality Assurance, and Performance Monitoring

Contents

1 Thrive@Pitt

The University of Pittsburgh's Thrive@Pitt initiative (`https://www.thrive.pitt.edu`) serves as a holistic platform for supporting student well-being and success. It centralizes mental health resources, physical wellness programs, academic support, financial guidance, and other valuable services to help students flourish during their time at the university. By bringing these resources together in one accessible location, Thrive@Pitt simplifies the process of connecting with the right support, fosters a culture of proactive well-being, and addresses the diverse needs of the Pitt community.

In practice, Thrive@Pitt:

- Creates a single digital portal where students, faculty, and staff can discover campus-wide well-being and support services.
- Encourages early intervention by highlighting resources before crises escalate, thereby bolstering student retention and mental health.
- Promotes an inclusive community culture that supports holistic development—academically, emotionally, and socially.
- Evolves continuously, adding new resources and tailoring its content based on community feedback and emerging wellness challenges.

2 Introduction

This project addresses the information accessibility challenge by building an end-to-end automated QA generation framework that produces high-quality question-answer pairs derived from the Thrive@Pitt website content. The system incorporates retrieval-augmented generation architecture, semantic document processing with entity recognition, enhanced web crawling with anti-bot detection, and multi-dimensional quality control—all optimized specifically for university support services. These question-answer pairs will be used to finetune and train SkillBuilder.io's chatbot model.

Building on the robust well-being ecosystem of Thrive@Pitt, the University of Pittsburgh sought to further enhance access to resources through an automated conversational assistant powered by SkillBuilder.io. While Thrive@Pitt already centralizes critical information, end-users often have specific, nuanced questions—ranging from "How do I apply for emergency funding?" to "What mental health services are available during midterms?" Enabling an AI-powered chatbot to answer such queries can bring immediate, user-friendly assistance to the community. However, constructing a chatbot that both understands complex domain-specific queries and provides accurate, context-rich responses is non-trivial. Careful data preparation and a robust question-answering (QA) pipeline are crucial for the chatbot's success. The quality of training data directly impacts model performance—better question-answer pairs lead to more accurate and helpful responses from the resulting chatbot, ultimately delivering more precise information about Thrive services, programs, and resources to the university community. A robust data pipeline must:

- **Extract relevant content** from Thrive@Pitt's pages using a specialized web crawler that respects site structure and avoids detection as a bot.
- **Segment and annotate content** into meaningful chunks for more precise retrieval and generation processes.
- **Generate realistic QA pairs** that cover the breadth of Thrive@Pitt's services, ensuring the chatbot can address varied queries.
- **Minimize hallucinations** and maintain factual consistency by leveraging advanced retrieval-augmented generation (RAG) workflows.
- **Adapt to new or updated content** on the Thrive@Pitt site so that the chatbot remains accurate over time.

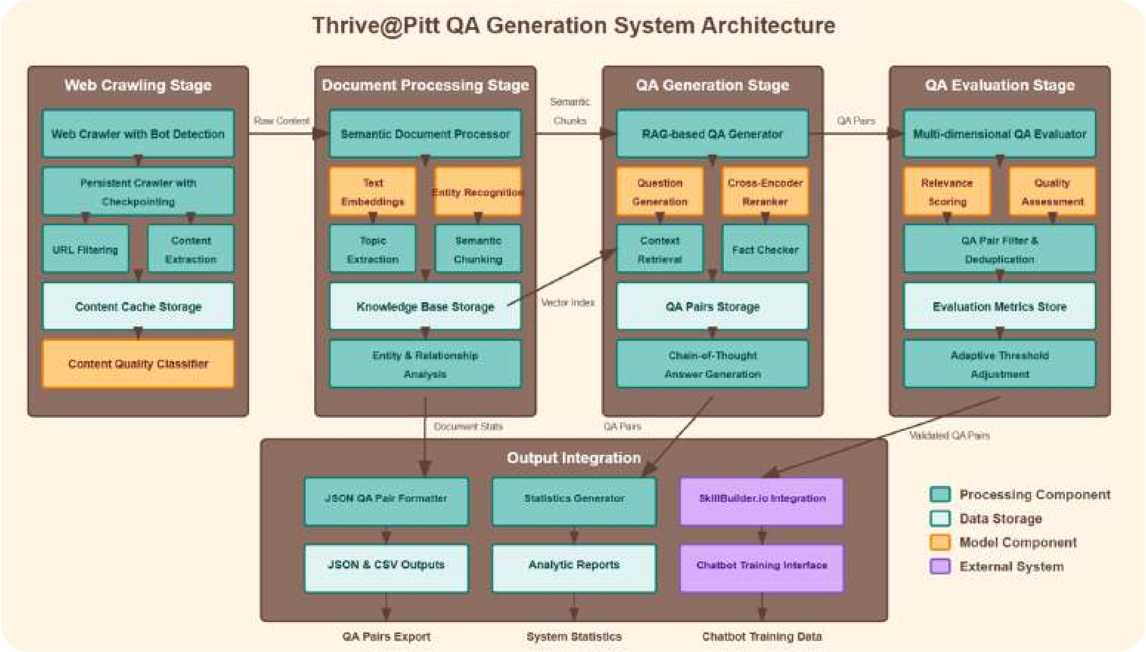

Figure 1: Thrive@Pitt QA Generation System Architecture. The diagram illustrates the complete pipeline flow across five main stages: (1) Web Crawling Stage with Bot Detection and Persistent Checkpointing; (2) Document Processing Stage with Entity Recognition and Semantic Chunking; (3) QA Generation Stage featuring RAG-based generation with Cross-Encoder Reranking and Fact Checking; (4) QA Evaluation Stage implementing Multi-dimensional Quality Assessment and Adaptive Thresholding; (5) Output Integration with formatters for JSON/CSV outputs and SkillBuilder.io integration. Color-coding distinguishes between Processing Components (teal), Data Storage (light teal), Model Components (orange), and External Systems (purple).

3 Project Overview

The project delivers an end-to-end automated application that employs a multi-stage approach to transform website content into curated question–answer (QA) pairs. Designed to support the University of Pittsburgh's websites, the system extracts and processes information, producing high-quality QA data optimized for training conversational agents. By integrating sophisticated AI framework with rigorous multi-dimensional evaluation metrics, the system ensures both the accuracy and practical relevance of the generated QA pairs, enhancing accessibility and engagement for users.

1. **Web Crawling:** A domain-optimized crawler systematically extracts textual content from Thrive@Pitt, ignoring duplicate or irrelevant assets and respecting robots.txt rules. To avoid bot-detection, it varies user-agent strings, adheres to realistic delays, and logs successful vs. failed attempts with checkpointing for resilience.

2. **Document Processing:** The extracted content is cleaned, normalized, and semantically chunked. Named entities and key phrases (e.g., organization names, student services) are identified to boost downstream retrieval accuracy.

3. **Retrieval-Augmented Generation (RAG):**

   - *Chunk-Level Embeddings:* Each text chunk is transformed into a vector representation, enabling semantic search and ranking.
   - *QA Generation:* An advanced language model is given a relevant subset of chunks and tasked with creating natural, domain-specific question–answer pairs.
   - *Consistency Checks:* The system employs cross-referencing techniques to confirm that generated answers match the retrieved text, reducing hallucinations and reinforcing factual correctness.

4. **Quality Evaluation and Filtering:** A multi-dimensional scoring method appraises each QA pair's relevance, completeness, factuality, and format. Lower-quality or near-duplicate pairs are discarded, ensuring the final dataset remains tightly aligned with Thrive@Pitt's official information.

5. **Output Integration:** Curated QA pairs, along with metadata (page source, timestamp, and quality metrics), are compiled into JSON or CSV formats. These can be ingested directly by chatbot frameworks (like SkillBuilder.io), enabling easy finetuning or augmentation of existing conversational models.

4 Key Benefits

1. **High-Fidelity Training Data.** By automatically extracting and verifying QA pairs from Thrive@Pitt content, the pipeline provides the chatbot with a rich, accurate dataset. This comprehensive coverage allows the chatbot to answer diverse user questions more reliably, without relying on manually curated or out-of-date information.

2. **Cross-Domain Applicability and Faster Deployment.** Beyond Thrive@Pitt, the QA generation system can be easily adapted to other university websites, rapidly producing curated QA data for diverse chatbot applications. This lowers the time and effort needed for manual content curation, accelerating the deployment of accurate, up-to-date support across different departmental or campus resources.

3. **Fewer Hallucinations and Errors.** The pipeline's cross-referencing and multi-step validation ensure that QA pairs match official site content. This reduces the risk of "hallucinated" or fabricated answers, improving the chatbot's trustworthiness and overall user satisfaction.

4. **Rapid Content Refresh and Adaptability.** Whenever Thrive@Pitt webpages are updated, the pipeline can re-crawl and re-process content. Newly extracted QA data is fed into the chatbot, ensuring it always reflects the latest well-being services, guidelines, and announcements.

5. **Modular Architecture for Future Enhancements.** The pipeline's design supports easy integration of newer advanced AI models or expanded retrieval techniques. This adaptability helps the chatbot remain state-of-the-art, whether adding advanced embeddings, cross-encoders, or fact-checking modules.

6. **Streamlined Data Integration and Maintenance.** Automated data cleaning, chunking, and QA generation minimize the need for manual oversight. The resulting QA pairs can be directly loaded into chatbot frameworks, reducing downtime and maintaining consistent, high-quality dialogues as new content appears.

7. **Improved Engagement and Resource Utilization.** By continually generating relevant, timely QA pairs, the pipeline equips the chatbot to direct users to the right services at the right time. This targeted guidance boosts user trust and satisfaction, encouraging more interactions with university wellness programs. In turn, it increases the overall utilization of resources by proactively connecting students, faculty, and staff with vital information before issues escalate.

5 System Architecture

The QA Generation System has a modular pipeline designed to automatically transform website content from any University of Pittsburgh website into a high-quality dataset of question-answer (QA) pairs. This dataset is specifically tailored for training conversational agents, to provide students with readily accessible information about university resources. Figure 1 provides a high-level overview of the system's architecture, illustrating the data flow and key components. The system comprises five primary stages: Web Crawling, Document Processing, QA Generation, QA Evaluation, and Output Integration. Each stage employs a combination of rule-based techniques and state-of-the-art neural models to ensure accuracy, relevance, and robustness.

5.1 Stage 1: Web Crawling

Goal: To collect relevant textual content from any university domain while adhering to ethical web crawling practices and mitigating the risk of bot detection.

This stage utilizes a custom-built web crawler, `PersistentCrawler`, which extends an `EnhancedWebCrawler` class. The crawler leverages libraries such as `requests` for HTTP communication and `BeautifulSoup` for HTML parsing. Key features include:

- **Bot Detection Avoidance:** The crawler employs several techniques to mimic human browsing behavior and avoid triggering anti-bot mechanisms:
  - ***User-Agent Rotation:*** A pool of modern user-agent strings (e.g., Chrome, Firefox, Safari) is randomly selected for each request.
  - ***Request Delays:*** Variable delays, following a distribution that simulates human reading times (implemented in `_get_random_delay()`), are introduced between requests. The base delay is configurable (defaulting to 2.0 seconds) and is further randomized.
  - ***Referrer Headers:*** Realistic `Referer` headers, mimicking navigation from previously visited pages, are included in requests.
  - ***Session Management:*** A `requests.Session` object with a persistent cookie jar (`cookiejar.LWPCookieJar`) is used to maintain state across requests, simulating a continuous browsing session.
  - ***HEAD Requests:*** Occasional HEAD requests are made before GET requests, mimicking browser behavior.
  - ***Robots.txt Respect:*** The crawler parses and adheres to the `robots.txt` file, respecting `Disallow` rules and `Crawl-delay` directives (implemented in `_check_robots_txt()`).
- **URL Filtering and Prioritization:** The crawler focuses on relevant content by:
  - ***Domain Restriction:*** Only URLs within the `thrive.pitt.edu` domain and its subdomains are crawled (implemented in `_is_valid_url()`).
  - *Content Type Filtering:* Non-HTML content (e.g., images, PDFs, documents) is ignored.
  - ***Priority Paths:*** A configurable list of URL path prefixes (`priority_paths`) allows prioritizing pages likely to contain valuable well-being information (e.g., "/services", "/resources").
  - ***URL Normalization:*** URLs are normalized to remove fragments and unnecessary query parameters, preventing redundant crawling.
- **Content Quality Assessment:** A heuristic scoring function (`_get_page_quality_score()`) evaluates the quality of each crawled page based on:
  - ***Content Length:*** Longer pages are generally favored, with penalties for very short pages.
  - ***Presence of Meaningful Phrases:*** The presence of domain-specific keywords and phrases related to well-being and university services (defined in `meaningful_phrases`) increases the score.
  - ***Structural Cues:*** The presence of HTML headings (detected during text extraction) is considered a positive indicator of structured content.
  - ***Bot Detection Patterns:*** The presence of text patterns indicative of bot detection or access denial (defined in `bot_patterns`) results in immediate rejection of the page.

  The quality score is a floating-point value between 0.0 and 1.0.
- **Checkpointing and Persistence:** The crawler's state is periodically saved to disk (implemented in `save_checkpoint()` and `load_checkpoint()`), allowing it to resume from interruptions without data loss. The checkpoint includes:
  - ***Visited URLs:*** A set of URLs that have been successfully crawled.
  - ***Failed URLs:*** A set of URLs that resulted in errors or were deemed invalid.
  - ***Content Cache:*** A dictionary mapping URLs to extracted and cleaned text content.
  - ***Page Scores:*** A dictionary mapping URLs to their calculated quality scores.

  Checkpoint files are stored in a configurable directory (`checkpoint_dir`, defaulting to "checkpoints") and are named using a unique identifier based on the base URL. A validation step (`validate_checkpoint()`) removes low-quality content from the cache upon loading.

The output of the Web Crawling stage is a *content cache*, a dictionary where keys are URLs and values are the extracted and cleaned text content of the corresponding pages. This cache, along with metadata such as quality scores and visit timestamps, serves as the input for the Document Processing stage.

5.2 Stage 2: Document Processing

Goal: To transform the raw, unstructured text extracted from web pages into a semantically enriched and structured representation suitable for downstream QA generation.

This stage employs a `DocumentProcessor` class, which utilizes a combination of rule-based techniques and neural models for text cleaning, semantic analysis, and chunking. The core data structures are `SemanticDocument` and `SemanticChunk`, representing entire web pages and coherent segments within them, respectively.

- **Text Normalization:** The raw text from each page is cleaned using the `_clean_text()` method. This involves:
  - ***Whitespace Handling:*** Excessive whitespace and newline characters are normalized.
  - ***Unicode Correction:*** Common Unicode character issues (e.g., incorrect apostrophes) are corrected.
  - ***HTML Artifact Removal:*** Remaining HTML entities and tags are removed.
  - ***Section Header Handling:*** Markdown-style section headers (e.g., "HEADING:", "TITLE:") are standardized.
- **Named Entity Recognition (NER):** The `analyze_entities()` method, managed by the `ResourceManager`, attempts to identify named entities (e.g., organizations, people, dates, locations) within the text. It prioritizes spaCy (`en_core_web_md` or `en_core_web_sm`) for high-quality NER. If spaCy is unavailable or fails, it falls back to NLTK's named entity chunker, and finally to a set of regular expressions for basic entity extraction. Extracted entities are stored with their text, label, and character offsets.
- **Key Phrase Extraction:** The `extract_key_phrases()` method, also managed by the `ResourceManager`, identifies important phrases and topics within the document. It primarily uses TF-IDF (Term Frequency-Inverse Document Frequency) to score n-grams (1-3 words). Stop words are removed, and a maximum number of features are considered. If TF-IDF fails, it falls back to extracting noun phrases using spaCy (if available) or, as a last resort, extracting capitalized phrases using regular expressions.
- **Semantic Chunking:** The `_create_semantic_chunks()` method divides the document into smaller, contextually coherent units (chunks). The chunking strategy adapts based on the document's structure:
  - ***Section-Based Chunking:*** If the document contains markdown-style section headers ("" or ""), it is split into chunks based on these headers (`_chunk_by_sections()`).
  - ***Paragraph-Based Chunking:*** If no section headers are found, the document is split into chunks based on paragraph breaks (""). A target chunk length (approximately 1000 characters) is maintained, with some overlap between adjacent chunks to preserve context (`_chunk_by_paragraphs()`).
  - ***Sentence-Based Chunking:*** If paragraph-based chunking results in very large chunks (greater than 1500 characters), a sentence-based splitting with overlap is applied (`_split_large_chunk()`). This uses a robust, fallback sentence tokenizer (`get_fallback_tokenize()`) that handles various edge cases and prioritizes NLTK, then spaCy, and finally a regex-based approach.

  Each `SemanticChunk` object stores the chunk's text, the source document URL, a unique index, and any extracted entities.
- **Embedding Generation:** Each chunk is converted into a dense vector representation (embedding) using a pre-trained sentence transformer model. The `ModelManager` class handles loading and unloading models to manage GPU memory efficiently. It prioritizes `all-mpnet-base-v2` for high-quality embeddings, falling back to smaller models (e.g., `all-MiniLM-L12-v2`, `all-MiniLM-L6-v2`) if necessary. The embedding dimension depends on the chosen model (e.g., 768 for `all-mpnet-base-v2`). Embeddings are stored as PyTorch tensors.

The output of the Document Processing stage is a `KnowledgeBase` object. This object stores:

- **`documents`:** A dictionary mapping URLs to `SemanticDocument` objects. Each `SemanticDocument` contains the full text, title, extracted entities, topics, and a list of `SemanticChunk` objects.

- **chunks:** A list of all SemanticChunk objects across all documents.
- **embeddings:** A PyTorch tensor containing the embeddings for all chunks, used for efficient similarity search.
- **chunk_text_index:** A simple text-based index mapping terms to chunk indices, used as a fallback for retrieval.

The KnowledgeBase provides methods for adding documents, rebuilding the search index, and performing semantic searches using cosine similarity between query embeddings and chunk embeddings. It also includes a fallback text-based search for cases where embeddings are unavailable.

5.3 Stage 3: QA Generation

Goal: To generate high-quality, contextually relevant question-answer pairs based on the processed documents and chunks in the knowledge base.

This stage utilizes a QAGenerator class, which implements a Retrieval-Augmented Generation (RAG) approach, combining neural language models with a retrieval mechanism to ground generated QA pairs in the source content.

- **Model Loading:** The QAGenerator loads several pre-trained models using the ModelManager:
  - ***Embedding Model:*** Used for retrieving relevant chunks (prioritizes all-mpnet-base-v2).
  - ***Question Generation Model:*** A T5-based model (Flan-T5-XL/XXL/Large/Base, with fallback to smaller models) used to generate questions based on context.
  - ***Answer Generation Model:*** A T5-based model (often the same as the question generation model) used to generate answers.
  - ***Fact Checker (Optional):*** A model (vectara/hallucination_evaluation_model) to assess the factual consistency of generated answers.
  - ***Cross-Encoder Re-ranker:*** A model (cross-encoder/ms-marco-MiniLM-L-12-v2 or cross-encoder/ms-marco-MiniLM-L-6-v2) to re-rank retrieved chunks based on their relevance to the question.
  - ***QA Evaluator (Optional):*** A model (cross-encoder/ms-marco-MiniLM-L-6-v2) to score the relevance of QA pairs.
- **Question Generation:** The generate_questions_from_documents() method generates questions for a given list of URLs. It prioritizes generating questions related to key topics extracted from the documents. For each topic, it calls generate_questions_for_topic(), which uses a combination of neural and rule-based approaches:
  - ***Neural Question Generation (_generate_questions_neural()):*** Uses the question generation model (Flan-T5) with a variety of prompts designed to elicit diverse and natural questions. Prompts are tailored based on the identified topic type (e.g., SERVICE, PROGRAM, LOCATION, WELLNESS). Chain-of-thought prompting is also used to encourage more sophisticated question generation.
  - ***Rule-Based Question Generation (_generate_questions_rule_based()):*** Used as a fallback if neural generation fails or produces insufficient questions. This method generates questions based on sentence patterns and topic types.
  - ***General Question Generation (generate_general_questions()):*** Generates general questions about the document, not tied to specific topics.
  - ***Adaptive Question Generation (_generate_adaptive_questions()):*** Used when very few questions are generated, providing more general questions that are less dependent on specific content.

  Generated questions are deduplicated (_deduplicate_questions()) and cleaned (_clean_question()).
- **Context Retrieval:** The _retrieve_context() method retrieves relevant chunks from the knowledge base for a given question. It uses a hybrid approach:

- *Vector Similarity Search:* Calculates the cosine similarity between the question embedding and the embeddings of all chunks in the knowledge base.
  - *Cross-Encoder Re-ranking:* The top-k retrieved chunks are re-ranked using a cross-encoder model, which provides a more accurate measure of relevance between the question and each chunk.
  - *Diversity Selection:* A greedy algorithm (`_select_diverse_chunks()`) selects a diverse set of chunks, prioritizing those from different source documents, to provide a broader context for answer generation.
  - *Fallback Text Search:* If vector search fails or returns too few results, a fallback text-based search (`_fallback_search()`) is used.

- **Answer Generation:** The `_generate_answer()` method generates an answer for a given question and its retrieved context. It uses the answer generation model (Flan-T5) with a detailed prompt that instructs the model to:
  - Synthesize information from multiple context chunks.
  - Maintain a confident and direct tone.
  - Avoid disclaimers ("I don't have enough information") if any relevant information is available.
  - Provide only information that is supported by the context.

  The method uses beam search for answer generation and selects the best answer from multiple candidates based on length and factual consistency. The `_merge_context_advanced()` method combines the retrieved chunks into a single context string, with clear markers indicating the source document of each chunk.

- **Fact Checking (Optional):** If a fact-checking model is available, it is used to assess the factual consistency of the generated answer with the retrieved context.

The output of the QA Generation stage is a list of `qa_pairs`, where each pair is a dictionary containing the question, answer, source URL, topic, topic type, and initial quality scores.

5.4 Stage 4: QA Evaluation

Goal: To assess the quality of the generated QA pairs and filter out low-quality or redundant pairs.

This stage evaluates each QA pair based on multiple criteria and assigns an overall quality score. The `QAGenerator` class's `_evaluate_answer()` method performs this evaluation.

- **Multi-Dimensional Scoring:** Each QA pair is scored based on the following metrics:
  - *Relevance (R):* Measures how well the answer aligns with the question. It combines a model-based relevance score (using a cross-encoder or the `qa_evaluator` model) with a keyword overlap score:
  $$R = 0.7 \times (\text{model-based relevance}) + 0.3 \times (\text{keyword match}).$$
  - *Factuality (F):* Assesses how faithfully the answer reflects the source text. It uses n-gram overlap between the answer and the retrieved context. An optional fact-checking model can override this score:
  $$F = 0.2 + 0.8 \times \frac{|\text{overlap n-grams}|}{|\text{answer n-grams}|}.$$
  - *Completeness (C):* Checks if the answer addresses all aspects of the question. Short or truncated answers receive lower scores. $C \in [0, 1]$.
  - *Formatting (Φ):* Ensures the answer is free of formatting issues, disclaimers, and repetitive text. Each major formatting violation lowers the score. $\Phi \in [0, 1]$.

- **Overall Score (O):** A weighted sum of the individual metrics:
$$O = 0.25R + 0.35F + 0.25C + 0.15\Phi.$$

- **Filtering:** QA pairs with an overall score below a threshold (defaulting to 0.5) are discarded. The threshold can be adjusted adaptively based on the overall quality distribution of the generated pairs.

- **Deduplication:** Near-duplicate QA pairs are identified and removed using a combination of Jaccard similarity (on word sets) and Levenshtein distance (edit distance). The `_questions_are_similar()` method implements this check. The `_deduplicate_questions()` method removes duplicate questions, and a similar approach is used for answers.

The `filter_qa_pairs()` method combines the scoring, filtering, and deduplication steps. It also includes an adaptive threshold adjustment, lowering the minimum score if very few QA pairs meet the initial threshold.

5.5 Stage 5: Output Integration

Goal: To package the filtered and evaluated QA pairs into various formats suitable for downstream use, such as training conversational AI agents.

This stage prepares the final QA dataset and provides analytics about its content and quality.

- **Data Export:** The QA pairs are exported in multiple formats:
  - ***JSON:*** A JSON file containing the QA pairs, along with metadata (source URL, topic, scores).
  - ***CSV:*** A CSV file for easy inspection and analysis in spreadsheet software.
  - ***Markdown:*** A Markdown file for human-readable documentation.
- **Analytics and Reporting:** The system generates summary statistics about the QA dataset, including:
  - ***Total QA Pairs:*** The number of QA pairs in the final dataset.
  - ***Average Scores:*** The average relevance, factuality, completeness, formatting, and overall scores.
  - ***Score Distribution:*** The number of QA pairs in different quality categories (e.g., excellent, good, average, below average).
  - ***Topic Coverage:*** The distribution of QA pairs across different topics.
  - ***URL Coverage:*** The distribution of QA pairs across different source URLs.
- **Chatbot Integration:** The generated QA pairs are designed to be directly integrated into conversational AI platforms, such as SkillBuilder.io. The JSON format is particularly suitable for this purpose. The data can also be used for fine-tuning retrieval-augmented generation (RAG) models or other language models.

The output of this stage is a set of files containing the curated QA pairs in various formats, along with a report summarizing the dataset's characteristics.

6 Technical Implementation

The Thrive@Pitt QA Generation System is implemented in Python, leveraging a range of state-of-the-art libraries and frameworks for natural language processing, web crawling, and machine learning. Key components include:

- **Core NLP Libraries:**
  - ***Hugging Face Transformers:*** Provides access to pre-trained language models (e.g., Flan-T5, BERT) and tokenizers.
  - ***spaCy:*** Used for named entity recognition (NER) and sentence boundary detection.
  - ***NLTK:*** Used for tokenization, stop word removal, and as a fallback for NER.
  - ***SentenceTransformers:*** Used for generating sentence and document embeddings.
  - ***BeautifulSoup:*** Used for parsing HTML content.
- **Web Crawling:**
  - ***requests:*** Used for making HTTP requests.
  - ***BeautifulSoup:*** Used for parsing HTML content.
  - ***urllib.parse:*** Used for URL parsing and manipulation.

- **QA Generation and Evaluation:**
  - ***Flan-T5 (XL/XXL/Large/Base):*** Used for question generation and answer generation.
  - ***Cross-Encoder Re-rankers (ms-marco-MiniLM-L-12-v2, ms-marco-MiniLM-L-6-v2):*** Used for re-ranking retrieved context chunks.
  - ***Fact Checker (vectara/hallucination_evaluation_model):*** Used for evaluating the factual consistency of answers.
- **Resource Management:**
  - ***Dynamic Model Loading/Unloading:*** The `ModelManager` class dynamically loads and unloads models based on memory availability, allowing the system to operate efficiently even with limited GPU resources.
  - ***Quantization:*** Support for 4-bit quantization (using `BitsAndBytesConfig`) is included to reduce the memory footprint of large models.
  - ***Fallback Mechanisms:*** The system includes fallback mechanisms for various components (e.g., using smaller models, alternative NLP libraries, rule-based methods) to ensure robustness.
- **Error Handling and Resilience:**
  - ***Comprehensive Exception Handling:*** `try-except` blocks are used throughout the pipeline to handle potential errors gracefully.
  - ***Automatic Checkpointing:*** The system periodically saves its state (crawled content, processed documents, generated QA pairs) to allow for recovery from interruptions.
  - ***Logging:*** Detailed logging is implemented using the Python `logging` module to track progress and diagnose issues.
  - ***Backoff Strategies:*** The `backoff` library is used to implement retry mechanisms for network requests, handling temporary network issues.

The system is designed to be modular and extensible, allowing for easy integration of new models, algorithms, and evaluation metrics.

7 Concerns

Below, the project identifies the primary concerns that require careful attention when deploying the chatbot platform. Each concern highlights key questions to pose to SkillBuilder.io, followed by an explanation of why it matters.

7.1 Hallucination Management

Key Questions

- How will the platform ensure that Large Language Model (LLM) responses remain grounded in curated data from university website?
- What mechanisms (e.g., citation or reference checks) exist to detect or prevent hallucinations *in real time*?

Why It Matters: Hallucinations occur when an LLM fabricates information not supported by its training data Ji et al. (2023); OpenAI (2023), posing a serious risk of misinformation about Thrive@Pitt services or unverified health recommendations. Ensuring the chatbot remains factually grounded preserves user trust and fosters engagement.

7.2 Data Updates and Exclusions

Key Questions

- How does the platform prevent outdated or invalid embeddings from influencing new responses after the model parameters have been fine-tuned on older information?

- Is there version control (or a similar mechanism) that tracks changes to the data over time?
- How does the platform incorporate new information—through fine-tuning or a RAG implementation—to update the knowledge base when outdated data is removed and replaced?

Why It Matters: University periodically updates webpages or retires outdated content. If the chatbot fails to synchronize with these changes, users could receive irrelevant or contradictory information. Proactive and transparent versioning ensures that new data supersedes the old, enabling reliable, current answers.

7.3 Toxicity Filtering / Adversarial questions / Guard Rails

Key Questions

- How does the platform handle incoming user inputs that may contain hate speech, harassment, or otherwise harmful text?
- How is training data filtered for toxicity before being used for fine-tuning the model?
- What remediation steps are triggered if content is flagged as toxic or unsafe?

Why It Matters: The inadvertent addition of toxic content to training data can influence the model's parameter updates, leading to undesired biases or harmful outputs. Also, chatbots sometimes receive abusive or harmful text from users. Without robust detection and filtering, the chatbot could inadvertently propagate or respond to toxic language in a way that conflicts with campus policy.

7.4 Performance Monitoring and Accountability

Key Questions

- What reporting tools are available to measure the chatbot's effectiveness (e.g., user satisfaction, resolved inquiries)?
- Can the platform log failed queries and flag them for manual review or data updates?

Why It Matters: Ongoing performance tracking is crucial for maintaining quality and relevance. By logging user interactions and identifying unanswered or poorly answered queries, administrators can continuously optimize the chatbot's data and responses. This iterative improvement process ensures the chatbot remains an accurate, up-to-date resource for students, staff and faculty.

8 Sample QA Pairs and Evaluation Results

To demonstrate the effectiveness of our QA generation system, this section presents a sample of 30 question-answer pairs extracted from the full dataset of more than 300 question-answer pairs. These examples showcase the system's ability to generate contextually relevant, factually accurate, and well-formed QA pairs across various aspects of University's resources and services.

8.1 Sample of the Generated QA Pairs

Question	**Answer**
What is the Wellness Concierge Program?	WCP is a new initiative that offers one-on-one meetings with trained Peer Wellness Coaches (PWC). PWCs are fellow students available to support you and help you navigate your wellness jouey. The program aims to foster a community where Pitt students can grow and thrive. What to Expect from the Wellness Concierge Program? Prior to meeting the first time, you'll engage in activities and discussions with your PWC to help identify personal wellness needs, interests, and goals.
What is the Campus Well-Being Consortium at Pitt?	The Campus Well-Being Consortium at Pitt is a collective of faculty and staff members, student leaders and community members working together to foster an environment where students can thrive. Thrive at Pitt was formed to advance our mission: To support and maintain all aspects of student well-being by promoting and improving accessibility to a range of resources relevant to our diverse student body.
What is the mission of the Campus Well-Being Consortium at Pitt?	The University of Pittsburgh is committed to creating a community where all students can thrive – both as students and beyond. In 2020, a dedicated group of campus community members joined together to create the Campus Well-Being Consortium at Pitt. Thrive at Pitt was formed to advance our mission: To support and maintain all aspects of student well-being by promoting and improving accessibility to a range of resources relevant to our diverse student body.
What are the factors that comprise a pillar of well-being?	Well-being is multidimensional, which means it encompasses numerous aspects (or pillars) of our life. Each pillar is comprised and impacted by many individual factors (e.g. sleep, diet, mindset, exercise, relationships, etc.). When we tend to these factors through daily routines, goal setting and practice, the results can be very satisfying and rewarding. When certain parts are left out, or neglected, the outcome is likely not as good. In exploring, supporting, or enhancing your own well-being it is important to become familiar with, and give proper attention to, each pillar.
What are the pillars of well-being?	Well-being is multidimensional, which means it encompasses numerous aspects (or pillars) of our life. Each pillar is comprised and impacted by many individual factors (e.g. sleep, diet, mindset, exercise, relationships, etc.). When we tend to these factors through daily routines, goal setting and practice, the results can be very satisfying and rewarding. When certain parts are left out, or neglected, the outcome is likely not as good. In exploring, supporting, or enhancing your own well-being it is important to become familiar with, and give proper attention to, each pillar.

Question	**Answer**
How many pillars of well-being are there?	Well-being is multidimensional, which means it encompasses numerous aspects (or pillars) of our life. Each pillar is comprised and impacted by many individual factors (e.g. sleep, diet, mindset, exercise, relationships, etc.). When we tend to these factors through daily routines, goal setting and practice, the results can be very satisfying and rewarding. When certain parts are left out, or neglected, the outcome is likely not as good. In exploring, supporting, or enhancing your own well-being it is important to become familiar with, and give proper attention to, each pillar.
What is the Counseling Center?	The mission of the University Counseling Center (UCC) is to provide high quality services and programs that foster the academic and personal development, as well as the psychological well-being, of Pitt's diverse student body. From outreach and prevention to crisis intervention, the UCC is dedicated to supporting and empowering students to maximize their college experience and realize their fullest potential. Drop-In Services Call 412-648-7930 Group Counseling Let's Talk Sessions are drop-in format that offer informal, confidential consultation with a clinician. Consultations are typically 10-15 minutes long and are offered on a first-come, first serve basis.
What is the mission of the Campus Well-Being Consortium?	The Campus Well-Being Consortium is a collective of faculty and staff members, student leaders and community members working together to foster an environment where students can thrive. Thrive at Pitt was formed to advance our mission: To support and maintain all aspects of student well-being by promoting and improving accessibility to a range of resources relevant to our diverse student body.
What is well-being?	Highlights What is Well-Being? Well-being can be defined as an optimal and dynamic state that allows people to achieve their full potential both as an individual and a member of the greater community. Lea about well-being. Build Your Foundation Each pillar of well-bearing serves as a foundational block to support your overall well- being. Together they create a solid foundation that allows you to thrive. Explore the pillars of wellbeing.
What is the Thrive at Pitt website?	The University of Pittsburgh is committed to creating a community where all students can thrive – both as students and beyond. Thrive at Pitt was formed to advance our mission: To support and maintain all aspects of student well-being by promoting and improving accessibility to a range of resources relevant to our diverse student body.

Question	**Answer**
What are some of the pillars that comprise the well-being?	Well-Being Well-being is multidimensional, which means it encompasses numerous aspects (or pillars) of our life. Each pillar is comprised and impacted by many individual factors (e.g. sleep, diet, mindset, exercise, relationships, etc.). When we tend to these factors through daily routines, goal setting and practice, the results can be very satisfying and rewarding. When certain parts are left out, or neglected, the outcome is likely not as good. In exploring, supporting, or enhancing your own well-being it is important to become familiar with, and give proper attention to, each pillar. Independently, every pillar plays an essential role for supporting our well-bearing. Collectively, they provide a solid foundation that accounts for and holds up your well-well-being.
What is the focus of the RISE program?	The Reaching Inside Your Soul for Excellence (RISE) program helps students get acclimated to the campus and help develop necessary skills needed to succeed at the University of Pittsburgh. Students in the program receive a mentor that will work with them to take responsibility of their future, make meaningful connections, and much more during their time at Pitt.
Where can I find the Office of Cross Cultural and Leadership Development?	The Office of Cross Cultural and Leadership Development (CCLD) is located on the 6th floor of the William Pitt Union and is a hub of campus activity. CCLD advises some of the most active student organizations on campus, provides leadership development opportunities, and offers dynamic programming.
What was the Campus Well-Being Consortium created?	The Campus Well-Being Consortium is a collective of faculty and staff members, student leaders and community members working together to foster an environment where students can thrive. Thrive at Pitt was formed to advance our mission: To support and maintain all aspects of student well-being.
What are the main factors that impact our well-being?	Well-being influences your ability to flourish and thrive. While we all come from various backgrounds, have different experiences, and face distinct challenges, leaing to improve and maintain the various areas of your well-being can help you overcome obstacles, achieve goals, and lead a meaningful and fulfilling life. Highlights What is Well-Being?
What are the individual pillars?	The pillars of well-being are comprised and impacted by many individual factors (e.g. sleep, diet, mindset, exercise, relationships, etc.). When we tend to these factors through daily routines, goal setting and practice, the results can be very satisfying and rewarding. When certain parts are left out, or neglected, the outcome is likely not as good. In exploring, supporting, or enhancing your own wellbeing it is important to become familiar with, and give proper attention to, each pillar.
What are the physical well being services?	Physical well-being is the knowledge and practice toward positive health conditions. It involves the adoption of proactive measures such as regular physical activity, good nutrition, utilization of preventative health care, and abstaining from harmful behaviors such as alcohol consumption, tobacco and drug use.

Question	**Answer**
What is the best version of yourself?	To thrive is to continually pursue the best version of yourself. Your well-being is impacted in many ways, and giving attention to each area is essential to ensuring your basic needs are met. In addition, continual focus on improving your well-bearing can lead to numerous positive benefits. For example, individuals who maintain higher levels of well- being are more likely to: Have a more positive outlook on life and overall better mental health Have greater confidence and higher levels.
How does well-being measure up to our lives?	Well-being is broad and multifaceted with correlations to numerous aspects of our life. It is impacted by many factors, both psychological and physical. These factors not only impact our satisfaction with life, they also influence our desire or ability to pursue and achieve our individual and collective capabilities. Thus, when we dedicate necessary time and effort to each area of well-being, we increase our chances for success, happiness, and a life defined as thriving.
What is the definition of well-being?	Well-being can be defined as an optimal and dynamic state that allows people to achieve their full potential both as an individual and a member of the greater community. Lea about well-being. Build Your Foundation Each pillar of well-bearing serves as a foundational block to support your overall well- being. Together they create a solid foundation that allows you to thrive. Explore the pillars of well being. Wellness Concierge Program Personalized, peer-led support through one-on-one meetings to assist you on your well- Being jouey. Some Inspiration My mission in life is not merely to survive, but to thrive; and to do so with some passion, some humor, and some style. Maya Angelou What Does It Mean to Thrive? To thrive is to continually pursue the best version of yourself. Through continued maintenance of your well being, you can build a Foundation to help you thrive in the face of challenges and become your best self!
What is the foundation for well-being?	Highlights Well-being is an optimal and dynamic state that allows people to achieve their full potential both as an individual and a member of the greater community. To thrive is to continually pursue the best version of yourself. Through continued maintenance of your well-being, you can build a foundation to help you thrive in.
What does pursuing the best version of yourself benefit you?	To thrive is to continually pursue the best version of yourself. Through continued maintenance of your well-being, you can build a foundation to help you thrive in the face of challenges and become your best self!
What does environmental stewardship encompass?	Environmental stewardship involves adopting and advocating for efforts that ensure responsible and efficient resource management while increasing access to clean air and water; nutritious food; and adequate clothing, shelter, and safety in both the built and natural environments at a variety of places and scales (e.g., home, school, work, community, city, country, world).

Question	**Answer**
What is the Office of Cross Cultural and Leadership Development?	The Office of Cross Cultural and Leadership Development (CCLD) is located on the 6th floor of the William Pitt Union and is a hub of campus activity. CCLD advises some of the most active student organizations on campus, provides leadership development opportunities, and offers dynamic programming in the areas of diversity and culture, social justice, and Frateity and Sorority Life.
How can you improve your spiritual well-being?	Well-being influences your ability to flourish and thrive. While we all come from various backgrounds, have different experiences, and face distinct challenges, leaing to improve and maintain the various areas of your well-being can help you overcome obstacles, achieve goals, and lead a meaningful and fulfilling life. Highlights What is Well-Being?
What are some tips for physical well-being?	Environmental well-being is the knowledge and practice toward positive health conditions. It involves the adoption of proactive measures such as regular physical activity, good nutrition, utilization of preventative health care, and abstaining from harmful behaviors such as alcohol consumption, tobacco and drug use. 1 5 Tips for Environmental Well-Being Engage in at least 150 minutes of physical activity each week. 2, 3, 4 Set and maintain a healthy dietary patte. 5 Abstain from using alcohol, tobacco, and other drugs/substances. 6 Get 7-8 hours of sleep each night. 7 Schedule routine check-ups with a primary care physician (PCP) and other doctors.
What are some ways to achieve physical well-being?	Well-being is broad and multifaceted with correlations to numerous aspects of our life. It is impacted by many factors, both psychological and physical. These factors not only impact our satisfaction with life, they also influence our desire or ability to pursue and achieve our individual and collective capabilities. Thus, when we dedicate necessary time and effort to each area of well-being, we increase our chances for success, happiness, and a life defined as thriving.
What is physical well-being?	Physical well-being is the knowledge and practice toward positive health conditions. It involves the adoption of proactive measures such as regular physical activity, good nutrition, utilization of preventative health care, and abstaining from harmful behaviors such as alcohol consumption, tobacco and drug use.

Question	Answer
What is the main idea of well-being?	Highlights What is Well-Being? Well-being can be defined as an optimal and dynamic state that allows people to achieve their full potential both as an individual and a member of the greater community. Lea about well-being. Build Your Foundation Each pillar of well-bearing serves as a foundational block to support your overall well- being. Together they create a solid foundation that allows you to thrive. Explore the pillars of well being. Wellness Concierge Program Personalized, peer-led support through one-on-one meetings to assist you on your well- well being jouey. Some Inspiration My mission in life is not merely to survive, but to thrive; and to do so with some passion, some humor, and some style. Maya Angelou What Does It Mean to Thrive? To thrive is to continually pursue the best version of yourself. Through continued maintenance of your well being, you can build a Foundation to help you thrive in the face of challenges and become your best self!

8.2 QA Pair Structure Explanation

To illustrate the structure of our QA pairs, let's examine an example from the dataset:

- **Question:** "What is the Wellness Concierge Program?"
- **Answer:** "WCP is a new initiative that offers one-on-one meetings with trained Peer Wellness Coaches (PWC). PWCs are fellow students available to support you and help you navigate your wellness jouey. The program aims to foster a community where Pitt students can grow and thrive. What to Expect from the Wellness Concierge Program? Prior to meeting the first time, you'll engage in activities and discussions with your PWC to help identify personal wellness needs, interests, and goals"
- **Source URL:** https://thrive.pitt.edu/wellness-concierge-program
- **Topic:** "wellness"
- **Topic Type:** "WELLNESS"
- **Scores:**
  - *Relevance:* 0.9997 - Indicates nearly perfect relevance of the answer to the question
  - *Factuality:* 0.9787 - Indicates high factual accuracy with respect to source content
  - *Completeness:* 0.8000 - Indicates good but not exhaustive coverage of all aspects
  - *Formatting:* 1.0000 - Indicates perfect formatting with no artifacts or issues
  - *Overall:* 0.9425 - The weighted combination of all metrics, showing excellent quality

This QA pair exemplifies the high standards achieved by our generation system, with strong relevance, factuality, and appropriate formatting. The answer directly addresses the question and provides valuable information about the Wellness Concierge Program's purpose and process.

8.3 Evaluation Metrics Analysis

The evaluation metrics output provide insights into the overall quality and characteristics of the generated QA pairs:

Metric	Value
Total QA Pairs	225

Table 2: Total QA Pairs Generated

Quality Metric	Average Score
Relevance	0.547
Factuality	0.957
Completeness	0.665
Formatting	1.000
Overall	0.788

Table 3: Average Quality Scores

Quality Tier	Count
Excellent (Overall ≥ 0.8)	125
Good ($0.7 \leq$ Overall < 0.8)	40
Average ($0.6 \leq$ Overall < 0.7)	58
Below Average (Overall < 0.6)	2

Table 4: Quality Distribution

- **Overall Quality:** This is a weighted combination of all metrics, reflecting the comprehensive quality of each QA pair. A higher overall score indicates a better-quality QA pair. We categorize pairs as "Excellent" (0.8+), "Good" (0.7-0.8), "Average" (0.6-0.7), or "Below Average" (<0.6).
- **Relevance:** This metric evaluates how directly the answer addresses the question posed. Higher relevance scores indicate answers that provide information specifically pertinent to the question, without extraneous or tangential content.
- **Factuality:** This measures how accurately the answer reflects information contained in the source material. High factuality scores indicate minimal hallucination or fabrication, with content faithfully representing the source material.
- **Completeness:** This assesses whether the answer covers all aspects of the question comprehensively. Higher completeness scores indicate answers that address multiple dimensions of the question without significant omissions.
- **Formatting:** This evaluates the structural quality of the answer, including paragraph breaks, lists, and other formatting elements. Perfect formatting scores indicate answers that are well-structured and easy to read.
- **Topic Coverage:** We analyze the distribution of topics among the QA pairs to ensure comprehensive coverage across key domains, including well-being concepts, university initiatives, and available resources.
- **URL Distribution:** This examines the spread of source URLs used to generate answers, ensuring content is drawn from diverse sections of the website.

These metrics collectively help us assess the system's strengths and identify areas for improvement.

9 Code

```
import requests
from bs4 import BeautifulSoup
import json
from typing import List, Dict, Tuple, Set, Any, Optional, Union, Generator, Callable
from transformers import (
    AutoTokenizer, AutoModelForSeq2SeqLM, AutoModelForQuestionAnswering,
    AutoModelForSequenceClassification, T5ForConditionalGeneration,
    pipeline, BitsAndBytesConfig
)
import torch
from torch.utils.data import Dataset, DataLoader
import nltk
from nltk.tokenize import sent_tokenize
from nltk.corpus import stopwords
```

```
import spacy
import logging
import os
import sys
import re
import gc
import time
from tqdm import tqdm
from datetime import datetime
import backoff
from concurrent.futures import ThreadPoolExecutor, as_completed
import hashlib
import traceback
import signal
from urllib.parse import urljoin, urlparse
import random
from http import cookiejar
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
from sentence_transformers import SentenceTransformer, CrossEncoder, util
import argparse
import copy
import numpy as np
from collections import Counter
from itertools import combinations
import math
import uuid

# Configure GPU usage
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Set up logging
logging.basicConfig(
    level=logging.INFO,
    format=’%(asctime)s - %(levelname)s - %(message)s’,
    handlers=[logging.FileHandler(’qa_generator.log’), logging.StreamHandler()]
)
logger = logging.getLogger(__name__)

#---------------------------------------------------------------------
# Resource Setup and Initialization
#---------------------------------------------------------------------

class ResourceManager:

    def __init__(self):
        self.nltk_available = False
        self.spacy_available = False
        self.spacy_model = None
        self.stopwords = set()
        self.nltk_data_dir = os.path.join(os.getcwd(), "nltk_data")

    def setup_nltk(self):
        try:
            # Create NLTK data directory if it doesn’t exist
            if not os.path.exists(self.nltk_data_dir):
                os.makedirs(self.nltk_data_dir)

            # Add to NLTK’s search path
            nltk.data.path.append(self.nltk_data_dir)

            # List of resources to download
            nltk_resources = [
                ’punkt_tab’,
                ’stopwords’,
                ’wordnet’,
                ’averaged_perceptron_tagger’,
                ’maxent_ne_chunker’,
                ’words’
            ]

            # Download resources that aren’t already available
            for resource in nltk_resources:
                try:
                    # Check if resource exists before downloading
                    try:
                        nltk.data.find(f'{resource}')
                        logger.info(f"NLTK resource already available: {resource}")
                    except LookupError:
                        # Download if not found
                        nltk.download(resource, download_dir=self.nltk_data_dir, quiet=True)
                        logger.info(f"NLTK resource downloaded: {resource}")
                except Exception as e:
                    logger.warning(f"Error downloading NLTK resource {resource}: {str(e)}")

            # Verify punkt is available for sentence tokenization
            try:
                nltk.data.find('tokenizers/punkt')
                self.nltk_tokenizer = nltk.data.load('tokenizers/punkt/english.pickle')
                logger.info("NLTK punkt tokenizer loaded successfully")
            except LookupError:
                logger.warning("Could not load punkt tokenizer, will use fallback methods")
                self.nltk_tokenizer = None

            # Initialize stopwords
            try:
                self.stopwords = set(stopwords.words('english'))
                logger.info("NLTK stopwords loaded successfully")
            except:
                # Fallback stopwords if NLTK fails
                self.stopwords = {"the", "a", "an", "in", "on", "at", "is", "are", "and", "to", "of", "for", "with"}
                logger.info("Using fallback stopwords")

            self.nltk_available = True
            logger.info("NLTK setup complete and working properly")

        except Exception as e:
            logger.error(f"Error setting up NLTK: {str(e)}")
            # Set up basic fallback stopwords
            self.stopwords = {"the", "a", "an", "in", "on", "at", "is", "are", "and", "to", "of", "for", "with"}
            logger.info("Using basic fallback stopwords due to NLTK setup error")
            self.nltk_available = False


    def setup_spacy(self):
        """Set up spaCy with fallbacks for different model availability."""
        try:
            # Try to load the English model - starting with larger one
            try:
                import spacy
                self.spacy_model = spacy.load("en_core_web_md")
                logger.info("Loaded spaCy medium model (en_core_web_md)")
            except:
                # Try smaller model
                try:
                    self.spacy_model = spacy.load("en_core_web_sm")
                    logger.info("Loaded spaCy small model (en_core_web_sm)")
                except:
                    # If not installed, try to download
                    logger.info("Downloading spaCy model")
                    os.system("python -m spacy download en_core_web_sm")
                    self.spacy_model = spacy.load("en_core_web_sm")
                    logger.info("Downloaded and loaded spaCy small model")

            # Test the model
            test_text = "The University of Pittsburgh offers student services."
            doc = self.spacy_model(test_text)
            entities = [ent.text for ent in doc.ents]
            if len(entities) > 0 or len(doc) > 0:
                self.spacy_available = True
                logger.info("spaCy setup complete and working properly")
            else:
                logger.warning("spaCy loaded but not functioning properly")

        except Exception as e:
            logger.error(f"Error setting up spaCy: {str(e)}")
            logger.info("NER functionality will be limited")

    def setup_huggingface_access(self):
        """Set up HuggingFace access with token."""
        try:
            from huggingface_hub import login

            # Try environment variable first
            token = os.environ.get("HUGGINGFACE_TOKEN")

            # If not available, use a default (should be replaced with user’s token)
            if not token:
                token = "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"

            if token:
                login(token=token)
                logger.info("Authenticated with HuggingFace Hub")
            else:
                logger.warning("No HuggingFace token found, some models may not be accessible")

        except Exception as e:
            logger.error(f"Error setting up HuggingFace access: {str(e)}")

    def get_fallback_tokenize(self):
        """Get a robust sentence tokenization function that works with or without NLTK."""
        def tokenize_text(text):
            if not text:
                return []

            # Method 1: Try using NLTK’s punkt tokenizer directly if available
            if self.nltk_available and hasattr(self, ’nltk_tokenizer’) and self.nltk_tokenizer:
                try:
                    return self.nltk_tokenizer.tokenize(text)
                except Exception as e:
                    logger.warning(f"NLTK tokenizer failed: {str(e)}")

            # Method 2: Try nltk.sent_tokenize which might work even if we couldn’t load punkt directly
            if self.nltk_available:
                try:
                    from nltk.tokenize import sent_tokenize
                    return sent_tokenize(text)
                except Exception as e:
                    logger.warning(f"NLTK sent_tokenize failed: {str(e)}")

            # Method 3: Try spaCy if available
            if self.spacy_available and self.spacy_model:
                try:
                    # Use spaCy’s sentence boundary detection
                    doc = self.spacy_model(text[:10000])  # Limit text length for performance
                    return [sent.text for sent in doc.sents]
                except Exception as e:
                    logger.warning(f"spaCy sentence tokenization failed: {str(e)}")

            # Method 4: Strong regex-based fallback approach
            logger.info("Using regex-based sentence tokenization as fallback")

            try:
                # Handle common abbreviations to prevent false splits
                text = re.sub(r’\b(Mr|Mrs|Ms|Dr|Prof|Inc|Ltd|Co|Sr|Jr|Ph\.D|M\.D|B\.A|M\.A|i\.e|e\.g)\.’,
                              lambda m: m.group(0).replace(’.’, ’<PERIOD>’), text)

                # Handle decimal numbers and URLs to prevent false splits
                text = re.sub(r'(\d+)\.(\d+)', r'\1<PERIOD>\2', text)  # Decimal numbers
                text = re.sub(r'(www\.)|(http\.)', r'\1<PERIOD>', text)  # URLs

                sentences = []

                # First split by punctuation + space + capital letter
                temp_sentences = re.split(r'(?<=[.!?])\s+(?=[A-Z])', text)

                # Then handle end-of-text punctuation in each segment
                for segment in temp_sentences:
                    # Split segments that might end with punctuation
                    end_splits = re.split(r'(?<=[.!?])$', segment)
                    sentences.extend([s for s in end_splits if s])

                # Clean up sentences
                clean_sentences = []
                for s in sentences:
                    if not s.strip():
                        continue

                    # Restore periods
                    s = s.replace('<PERIOD>', '.')

                    # Add ending punctuation if missing
                    if not re.search(r'[.!?]$', s):
                        s = s + '.'

                    clean_sentences.append(s)

                # If we still have no sentences, try a simpler approach - split on paragraph breaks
                if not clean_sentences and '\n\n' in text:
                    paragraphs = text.split('\n\n')
                    for p in paragraphs:
                        if p.strip():
                            clean_sentences.append(p.strip())

                # If all else fails, treat the whole text as one sentence
                if not clean_sentences and text.strip():
                    clean_sentences = [text.strip()]

                return clean_sentences

            except Exception as e:
                logger.warning(f"Regex tokenization failed: {str(e)}")

                # Absolute last resort: simple period splitting
                try:
                    return [s.strip() + '.' for s in text.split('. ') if s.strip()]
                except:
                    # If everything fails, return original text as a single sentence
                    return [text] if text else []

        return tokenize_text

    def analyze_entities(self, text):
        """Extract named entities from text using available NER tools."""
        entities = []

        # Try spaCy first (best quality)
        if self.spacy_available and self.spacy_model:
            try:
                doc = self.spacy_model(text[:5000])  # Limit length for performance
                for ent in doc.ents:
                    entities.append({
                        "text": ent.text,
                        "label": ent.label_,
                        "start": ent.start_char,
                        "end": ent.end_char
                    })
            except Exception as e:
                logger.warning(f"spaCy entity extraction failed: {str(e)}")

        # If no entities found or spaCy not available, try NLTK
        if not entities and self.nltk_available:
            try:
                from nltk import word_tokenize, pos_tag, ne_chunk
                from nltk.chunk import tree2conlltags

                # Process with NLTK NER
                tokens = word_tokenize(text[:3000])  # Limit length
                pos_tags = pos_tag(tokens)
                ne_tree = ne_chunk(pos_tags)

                # Extract named entities
                iob_tags = tree2conlltags(ne_tree)
                current_entity = {"text": "", "label": "", "start": 0}
                char_index = 0

                for word, pos, tag in iob_tags:
                    if tag != "O":  # Part of a named entity
                        entity_label = tag.split("-")[1]

                        if tag.startswith("B-"):  # Beginning of entity
                            # Save previous entity if exists
                            if current_entity["text"]:
                                entities.append(current_entity.copy())

                            # Start new entity
                            current_entity = {
                                "text": word,
                                "label": entity_label,
                                "start": char_index
                            }
                        elif tag.startswith("I-"):  # Continuation of entity
                            current_entity["text"] += " " + word
                    else:
                        # End of entity
                        if current_entity["text"]:
                            entities.append(current_entity.copy())
                            current_entity = {"text": "", "label": "", "start": 0}

                    # Update character index (approximate)
                    char_index += len(word) + 1

                # Add final entity if exists
                if current_entity["text"]:
                    entities.append(current_entity)

            except Exception as e:
                logger.warning(f"NLTK entity extraction failed: {str(e)}")

        # If still no entities, use regex patterns for basic extraction
        if not entities:
            # Simple patterns for common entity types
            patterns = [
                (r’\b[A-Z][a-z]+ (University|College|School)\b’, ’ORG’),  # Educational institutions
                (r’\b[A-Z][a-z]+ (Center|Service|Office|Department)\b’, ’ORG’),  # University services
                (r’\b[A-Z][a-z]+ (Hall|Building|Library|Center)\b’, ’FAC’),  # Campus facilities
                (r’\b(January|February|March|April|May|June|July|August|September|October|November|December) \d{1,2}(st|nd|rd|th)?,? \d{4}\b’, ’DATE’),  # Dates
                (r’\b\d{1,2}:\d{2} (AM|PM|am|pm)\b’, ’TIME’),  # Times
                (r’\b[A-Z][a-z]+ [A-Z][a-z]+\b’, ’PERSON’)  # Potential names
            ]

            for pattern, label in patterns:
                for match in re.finditer(pattern, text):
                    entities.append({
                        "text": match.group(0),
                        "label": label,
                        "start": match.start(),
                        "end": match.end()
```

```python
                })

    return entities

def extract_key_phrases(self, text, top_n=10):
    """Extract key phrases that represent important concepts in the text."""
    if not text or len(text) < 20:
        return []

    # Clean text
    text = re.sub(r'\s+', ' ', text).strip()

    # Split into sentences
    tokenize = self.get_fallback_tokenize()
    sentences = tokenize(text)

    if not sentences:
        return []

    try:
        # Use TF-IDF to find important n-grams
        vectorizer = TfidfVectorizer(
            ngram_range=(1, 3),
            stop_words=list(self.stopwords) if self.stopwords else 'english',
            max_features=100
        )

        # Get matrix
        tfidf_matrix = vectorizer.fit_transform(sentences)

        # Get feature names
        feature_names = vectorizer.get_feature_names_out()

        # Calculate scores for each feature across all sentences
        feature_scores = tfidf_matrix.sum(axis=0).A1

        # Sort features by score
        sorted_features = sorted(zip(feature_names, feature_scores), key=lambda x: x[1], reverse=True)

        # Filter out single stop words and very short phrases
        key_phrases = []
        for phrase, score in sorted_features:
            if len(phrase) > 3 and not (len(phrase.split()) == 1 and phrase in self.stopwords):
                key_phrases.append({
                    "text": phrase,
                    "score": float(score),
                    "type": "PHRASE"
                })

                if len(key_phrases) >= top_n:
                    break

        return key_phrases

    except Exception as e:
        logger.error(f"Error extracting key phrases: {str(e)}")

        # Fallback: extract noun phrases if available
        if self.spacy_available and self.spacy_model:
            try:
                doc = self.spacy_model(text[:5000])
                noun_phrases = [{"text": chunk.text, "score": 0.5, "type": "NOUN_PHRASE"}
                               for chunk in doc.noun_chunks]
                return noun_phrases[:top_n]
            except:
                pass

        # Last resort: just return capitalized phrases
        cap_phrases = re.findall(r'\b[A-Z][a-zA-Z]*(?:\s+[A-Z][a-zA-Z]*)+\b', text)
        return [{"text": phrase, "score": 0.5, "type": "CAP_PHRASE"}
```

```
                   for phrase in set(cap_phrases)][:top_n]

#----------------------------------------------------------------------
# Enhanced Model Manager
#----------------------------------------------------------------------

class ModelManager:
    """Model manager with dynamic loading/unloading and quantization support."""

    def __init__(self, use_gpu: bool = True, memory_threshold: float = 0.95):
        """Initialize model manager with memory management."""
        self.use_gpu = use_gpu
        self.memory_threshold = memory_threshold
        self.device = torch.device("cuda" if use_gpu and torch.cuda.is_available() else "cpu")
        self.loaded_models = {}

        # Track how many models we have memory for
        if self.use_gpu and torch.cuda.is_available():
            self.total_gpu_memory = torch.cuda.get_device_properties(0).total_memory
            logger.info(f"GPU memory: {self.total_gpu_memory / 1e9:.2f} GB")

            available_memory = self.total_gpu_memory * self.memory_threshold
            self.max_models = max(2, min(10, int(available_memory / 1.5e9)))
            logger.info(f"Estimated capacity: {self.max_models} models can be loaded simultaneously")
        else:
            self.max_models = 2  # Conservative default for CPU

    def _check_memory(self):
        """Check if we have enough GPU memory available."""
        if not self.use_gpu or not torch.cuda.is_available():
            return True

        # Get current memory usage
        allocated = torch.cuda.memory_allocated()
        reserved = torch.cuda.memory_reserved()
        total = self.total_gpu_memory

        # Calculate percentage used
        used_fraction = (allocated + reserved) / total

        # Log warning if close to threshold
        if used_fraction > self.memory_threshold * 0.8:
            logger.warning(f"GPU memory usage high: {used_fraction:.1%}")

        # Return True if we have enough memory
        return used_fraction < self.memory_threshold

    def load_model(self, model_name: str, model_class, model_path: str, **kwargs):
        """Load a model with intelligent memory management and quantization."""
        # If model already loaded, return it
        if model_name in self.loaded_models:
            logger.info(f"Model {model_name} already loaded")
            # Update last used timestamp
            self.loaded_models[model_name]['last_used'] = time.time()
            return self.loaded_models[model_name]['model']

        # Check if we need to unload models to free memory
        if len(self.loaded_models) >= self.max_models:
            logger.info(f"Maximum models loaded ({len(self.loaded_models)}), unloading least recently used")
            self._unload_least_used()

        # Check if we have enough memory
        if not self._check_memory():
            logger.warning("Memory usage high, forcing garbage collection")
            gc.collect()
            if self.use_gpu and torch.cuda.is_available():
                torch.cuda.empty_cache()

            # Check again after cleaning
            if not self._check_memory():
```

```
            # Unload more aggressively
            if self.loaded_models:
                logger.warning("Still low on memory, unloading all models")
                self.cleanup()
            else:
                logger.error("Not enough memory to load model even after cleanup")
                raise MemoryError("Not enough GPU memory available")

    # Apply quantization if requested
    quantization_config = None
    if kwargs.pop('quantize', False) and self.use_gpu:
        quantization_config = BitsAndBytesConfig(
            load_in_4bit=True,
            bnb_4bit_use_double_quant=True,
            bnb_4bit_quant_type="nf4",
            bnb_4bit_compute_dtype=torch.float16
        )
        logger.info(f"Using 4-bit quantization for {model_name}")

    # Try to load the model with retries for network issues
    @backoff.on_exception(backoff.expo,
                         (requests.RequestException, OSError),
                         max_tries=3, max_time=60)
    def load_with_retry():
        if model_class == SentenceTransformer:
            # Special handling for sentence transformers
            model = SentenceTransformer(model_path, device=self.device)
        else:
            # Default loading with quantization if specified
            if quantization_config:
                model = model_class.from_pretrained(
                    model_path,
                    quantization_config=quantization_config,
                    device_map="auto" if self.use_gpu else None,
                    **kwargs
                )
            else:
                # Handle device placement
                if 'device_map' not in kwargs and self.use_gpu:
                    if hasattr(model_class, 'from_pretrained') and 'auto' in dir(model_class):
                        # This model supports auto device mapping
                        kwargs['device_map'] = "auto"
                        model = model_class.from_pretrained(model_path, **kwargs)
                    else:
                        # Manual device placement
                        model = model_class.from_pretrained(model_path, **kwargs).to(self.device)
                else:
                    model = model_class.from_pretrained(model_path, **kwargs)

                    # Explicitly move to device if not using auto mapping
                    if 'device_map' not in kwargs and self.use_gpu:
                        try:
                            model = model.to(self.device)
                        except Exception as e:
                            logger.warning(f"Could not move model to {self.device}: {str(e)}")

        return model

    # Try to load the model
    try:
        logger.info(f"Loading model {model_name} from {model_path}")
        model = load_with_retry()
        self.loaded_models[model_name] = {
            'model': model,
            'last_used': time.time(),
            'size': self._estimate_model_size(model)
        }
        logger.info(f"Successfully loaded {model_name}")
        return model
    except Exception as e:
        logger.error(f"Error loading model {model_name}: {str(e)}")

        # Try smaller fallback models if primary ones fail
        if model_path == "google/flan-t5-xl":
            logger.info("Trying smaller T5 model instead")
            return self.load_model(model_name, model_class, "google/flan-t5-large",
**kwargs)
        elif model_path == "google/flan-t5-large":
            logger.info("Trying even smaller T5 model")
            return self.load_model(model_name, model_class, "google/flan-t5-base",
**kwargs)
        elif "v3-large" in model_path or "-large-" in model_path:
            logger.info("Trying base model instead of large")
            smaller_path = model_path.replace("large", "base")
            return self.load_model(model_name, model_class, smaller_path, **kwargs)

        raise

def _estimate_model_size(self, model):
    """Estimate the size of a model in bytes (rough approximation)."""
    try:
        # Get model parameters
        params = sum(p.numel() for p in model.parameters())

        # Estimate bytes per parameter
        bytes_per_param = 4  # Conservative estimate
        if hasattr(model, ’dtype’):
            if model.dtype in [torch.float16, torch.bfloat16]:
                bytes_per_param = 2
            elif model.dtype == torch.int8:
                bytes_per_param = 1

        # Return estimated size
        return params * bytes_per_param
    except:
        # If we can’t estimate, use a default value
        return 5e8

def _unload_least_used(self):
    """Unload the least recently used model."""
    if not self.loaded_models:
        return

    # Find least recently used
    lru_model = min(self.loaded_models.items(), key=lambda x: x[1][’last_used’])
    model_name = lru_model[0]

    # Unload it
    self.unload_model(model_name)

def unload_model(self, model_name: str):
    """Unload a specific model from memory."""
    if model_name not in self.loaded_models:
        return

    logger.info(f"Unloading model {model_name}")

    try:
        # Get model
        model = self.loaded_models[model_name][’model’]

        # Move to CPU first to free GPU memory
        if self.use_gpu:
            try:
                model = model.to(’cpu’)
            except:
                pass

        # Delete model
        del model

        # Remove from loaded models
        del self.loaded_models[model_name]
```

```python

            # Force garbage collection
            gc.collect()
            if self.use_gpu and torch.cuda.is_available():
                torch.cuda.empty_cache()

            logger.info(f"Successfully unloaded {model_name}")

        except Exception as e:
            logger.error(f"Error unloading model {model_name}: {str(e)}")

    def cleanup(self):
        """Unload all models and free memory."""
        if not self.loaded_models:
            return

        logger.info(f"Cleaning up, unloading {len(self.loaded_models)} models")

        # Get all model names first
        model_names = list(self.loaded_models.keys())

        # Unload each model
        for model_name in model_names:
            self.unload_model(model_name)

        # Final garbage collection
        gc.collect()
        if self.use_gpu and torch.cuda.is_available():
            torch.cuda.empty_cache()

        logger.info("All models unloaded")

#---------------------------------------------------------------------
# Enhanced Crawler with Anti-Bot Detection and Content Quality Filters
#---------------------------------------------------------------------

class EnhancedWebCrawler:
    """
    Web crawler with anti-bot detection avoidance, content quality
    assessment, and intelligent page prioritization.
    """

    def __init__(self, base_url: str, delay: float = 2.0, checkpoint_dir: str = "checkpoints"):
        """Initialize the crawler with robust settings."""
        self.base_url = base_url
        self.base_domain = urlparse(base_url).netloc
        self.visited_urls = set()
        self.failed_urls = set()
        self.delay = delay
        self.content_cache = {}
        self.checkpoint_dir = checkpoint_dir

        # Create checkpoint directory if it doesn't exist
        os.makedirs(checkpoint_dir, exist_ok=True)

        # Create a unique ID for this crawler instance based on the base URL
        self.checkpoint_id = hashlib.md5(base_url.encode('utf-8')).hexdigest()

        # For tracking page quality
        self.page_scores = {}

        # Create persistent session with cookies
        self.session = requests.Session()
        self.session.cookies = cookiejar.LWPCookieJar()

        # Rotate user agents to avoid detection - using more modern user agents
        self.user_agents = [
            'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/120.0.0.0 Safari/537.36',
            'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/16.5 Safari/605.1.15',
            'Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:120.0) Gecko/20100101 Firefox/120.0',
            'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/120.0.0.0 Safari/537.36 Edg/120.0.0.0',
            'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/120.0.0.0 Safari/537.36',
            'Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/120.0.0.0 Safari/537.36',
            'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/120.0.0.0 Safari/537.36 OPR/106.0.0.0',
            'Mozilla/5.0 (iPad; CPU OS 16_5 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/16.5 Mobile/15E148 Safari/604.1'
        ]

        # Headers to mimic real browsers
        self.browser_headers = {
            'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7',
            'Accept-Language': 'en-US,en;q=0.9',
            'Accept-Encoding': 'gzip, deflate, br',
            'Connection': 'keep-alive',
            'Upgrade-Insecure-Requests': '1',
            'Sec-Fetch-Dest': 'document',
            'Sec-Fetch-Mode': 'navigate',
            'Sec-Fetch-Site': 'none',
            'Sec-Fetch-User': '?1',
            'sec-ch-ua': '"Not_A Brand";v="99", "Google Chrome";v="109", "Chromium";v="109"',
            'sec-ch-ua-mobile': '?0',
            'sec-ch-ua-platform': '"Windows"'
        }

        # Actual bot detection patterns - much more focused to avoid false positives
        self.bot_patterns = [
            r'captcha\s+verification',
            r'security\s+check\s+failed',
            r'automated\s+access\s+detected',
            r'access\s+denied.*automated',
            r'blocked.*bot',
            r'cloudflare.*ray\s+id',
            r'your\s+IP\s+has\s+been\s+blocked'
        ]
        self.bot_regex = re.compile('|'.join(self.bot_patterns), re.IGNORECASE)

        # Content quality indicators
        self.meaningful_phrases = [
            # University/education terms
            'university', 'campus', 'college', 'academic', 'student', 'faculty', 'staff',
            'course', 'class', 'program', 'degree', 'major', 'minor', 'education',
            'research', 'study', 'learning',

            # Thrive/wellness specific terms
            'wellness', 'health', 'counseling', 'support', 'resource', 'service',
            'assistance', 'help', 'aid', 'benefit', 'initiative', 'thrive', 'success',
            'wellbeing', 'mental health', 'physical health', 'emotional health',

            # Academic support terms
            'advising', 'tutoring', 'mentoring', 'center', 'office', 'department',
            'financial aid', 'scholarship', 'grant', 'funding', 'career', 'job',
            'internship', 'opportunity'
        ]

        # Paths to prioritize - will be populated by caller
        self.priority_paths = []

        # Cache for robots.txt rules
        self.robots_rules = None
        self.crawl_delay = 0

        # Keep track of visit times to ensure natural browsing patterns
        self.last_visit_time = {}
        self.global_last_visit = time.time() - self.delay
```

```python
    def _get_checkpoint_path(self, file_type: str) -> str:
        """Get the path for a specific checkpoint file."""
        return os.path.join(self.checkpoint_dir, f"{file_type}_{self.checkpoint_id}.json")

    def save_checkpoint(self) -> None:
        """Save crawler state to checkpoint files."""
        try:
            # Save content cache
            cache_path = self._get_checkpoint_path("content")
            with open(cache_path, 'w', encoding='utf-8') as f:
                json.dump(self.content_cache, f)

            # Save visited URLs
            visited_path = self._get_checkpoint_path("visited")
            with open(visited_path, 'w', encoding='utf-8') as f:
                json.dump(list(self.visited_urls), f)

            # Save failed URLs
            failed_path = self._get_checkpoint_path("failed")
            with open(failed_path, 'w', encoding='utf-8') as f:
                json.dump(list(self.failed_urls), f)

            # Save page scores
            scores_path = self._get_checkpoint_path("scores")
            with open(scores_path, 'w', encoding='utf-8') as f:
                json.dump(self.page_scores, f)

            logger.info(f"Checkpoint saved: {len(self.content_cache)} pages")

        except Exception as e:
            logger.error(f"Error saving checkpoint: {str(e)}")

    def load_checkpoint(self) -> bool:
        """Load crawler state from checkpoint files."""
        try:
            # Check if checkpoint files exist
            cache_path = self._get_checkpoint_path("content")
            visited_path = self._get_checkpoint_path("visited")
            failed_path = self._get_checkpoint_path("failed")
            scores_path = self._get_checkpoint_path("scores")

            if not all(os.path.exists(path) for path in [cache_path, visited_path, failed_path]):
                logger.info("Incomplete checkpoint files, starting fresh")
                return False

            # Load content cache
            with open(cache_path, 'r', encoding='utf-8') as f:
                self.content_cache = json.load(f)

            # Load visited URLs
            with open(visited_path, 'r', encoding='utf-8') as f:
                self.visited_urls = set(json.load(f))

            # Load failed URLs
            with open(failed_path, 'r', encoding='utf-8') as f:
                self.failed_urls = set(json.load(f))

            # Load page scores if available
            if os.path.exists(scores_path):
                with open(scores_path, 'r', encoding='utf-8') as f:
                    loaded_scores = json.load(f)
                    # Convert string keys back to float values
                    self.page_scores = {k: float(v) if isinstance(v, str) else v
                                      for k, v in loaded_scores.items()}

            logger.info(f"Checkpoint loaded: {len(self.content_cache)} pages, {len(self.visited_urls)} visited URLs")
            return True

        except Exception as e:
            logger.error(f"Error loading checkpoint: {str(e)}")
            return False

    def validate_checkpoint(self) -> None:
        """Validate and clean checkpoint data."""
        if not self.content_cache:
            return

        # Check cached content for quality
        invalid_urls = []

        for url, content in tqdm(self.content_cache.items(), desc="Validating checkpoint"):
            if not content or not self._is_valid_content(content, url):
                invalid_urls.append(url)

        # Remove invalid content
        for url in invalid_urls:
            logger.warning(f"Removing invalid content for {url}")
            del self.content_cache[url]
            if url in self.page_scores:
                del self.page_scores[url]

        if invalid_urls:
            logger.info(f"Removed {len(invalid_urls)} invalid pages from checkpoint")

            # Save updated checkpoint
            self.save_checkpoint()

    def _is_valid_content(self, content: str, url: str) -> bool:
        """Check if content is valid (not an error page, has meaningful content)."""
        # Check for empty content
        if not content or len(content) < 50:  # Reduced minimum length
            logger.debug(f"Content too short for {url}")
            return False

        # Check only for definitive bot detection patterns
        if self.bot_regex.search(content):
            logger.warning(f"Bot detection pattern found for {url}")
            return False

        # Check if content has some basic text
        word_count = len(re.findall(r'\b\w+\b', content))
        if word_count < 10:
            logger.debug(f"Too few words ({word_count}) in {url}")
            return False

        return True

    def _check_robots_txt(self):
        """Parse robots.txt to respect crawling guidelines."""
        if self.robots_rules is not None:
            return

        try:
            # Get the robots.txt file
            robots_url = urljoin(self.base_url, "/robots.txt")
            logger.info(f"Checking robots.txt at {robots_url}")

            # Use a simple GET request with a short timeout
            user_agent = random.choice(self.user_agents)
            robots_response = requests.get(
                robots_url,
                timeout=5,
                headers={
                    'User-Agent': user_agent,
                    'Accept': 'text/plain,text/html;q=0.9,*/*;q=0.8'
                }
            )

            if robots_response.status_code == 200:
                # Parse robots.txt content
                content = robots_response.text
```

```python
            self.robots_rules = []

            # Simple parsing of disallow rules and crawl-delay
            current_agent = None
            for line in content.split('\n'):
                line = line.strip()

                # Skip empty lines and comments
                if not line or line.startswith('#'):
                    continue

                # Check for User-agent directive
                if line.lower().startswith('user-agent:'):
                    agent = line.split(':', 1)[1].strip().lower()
                    current_agent = agent

                # Check for Disallow directive for relevant user agents
                elif line.lower().startswith('disallow:') and (current_agent in ['*', 'bot', '']):
                    path = line.split(':', 1)[1].strip()
                    if path:
                        self.robots_rules.append(path)

                # Check for Crawl-delay directive for relevant user agents
                elif line.lower().startswith('crawl-delay:') and (current_agent in ['*', 'bot', '']):
                    try:
                        delay = float(line.split(':', 1)[1].strip())
                        # Use the largest crawl-delay
                        self.crawl_delay = max(self.crawl_delay, delay)
                    except ValueError:
                        pass

            logger.info(f"Found {len(self.robots_rules)} disallow rules in robots.txt")
            if self.crawl_delay > 0:
                logger.info(f"Found crawl-delay: {self.crawl_delay} seconds")
                # Update delay
                self.delay = max(self.delay, self.crawl_delay)
        else:
            logger.info(f"No robots.txt found or accessible, status: {robots_response.status_code}")
            self.robots_rules = []

    except Exception as e:
        logger.warning(f"Error parsing robots.txt: {str(e)}")
        self.robots_rules = []  # Continue without rules

def _is_valid_url(self, url: str) -> bool:
    """Check if URL is valid and should be crawled."""
    # Parse URL
    try:
        parsed = urlparse(url)
    except:
        return False

    # Basic URL validation
    if not all([parsed.scheme, parsed.netloc]) or parsed.scheme not in ['http', 'https']:
        return False

    # Check domain
    url_domain = parsed.netloc.lower()
    if not (url_domain == self.base_domain or
           url_domain.endswith('.' + self.base_domain) or
           self.base_domain.endswith('.' + url_domain)):
        return False

    # Check file extension - skip media, documents, etc.
    path = parsed.path.lower()
    if re.search(r'\.(jpg|jpeg|png|gif|pdf|doc|docx|ppt|pptx|zip|rar|exe|css|js|xml|json)$', path):
        return False

    # Skip common utility paths
    if re.search(r'/(login|logout|signin|signout|register|profile|cart|checkout|search\?)', path):
        return False

    # Check robots.txt rules
    if self.robots_rules:
        for rule in self.robots_rules:
            if path.startswith(rule):
                return False

    return True

def _get_random_delay(self):
    """Get a random delay to mimic human browsing patterns."""

    # Base delay factors - more variance
    delay_options = [
        self.delay * random.uniform(0.7, 1.0),
        self.delay * random.uniform(1.0, 1.5),
        self.delay * random.uniform(1.5, 3.0)
    ]

    # Choose a delay with weighting toward the middle
    weights = [0.3, 0.4, 0.3]
    return random.choices(delay_options, weights=weights)[0]

@backoff.on_exception(
    backoff.expo,
    (requests.RequestException, requests.exceptions.Timeout, requests.exceptions.ConnectionError),
    max_tries=3,
    giveup=lambda e: isinstance(e, requests.exceptions.HTTPError) and e.response.status_code in [403, 404, 405, 410],
)
def _get_page_content(self, url: str) -> Tuple[str, List[str]]:
    """Get content and links from a page with anti-bot detection measures."""

    current_time = time.time()

    # Global delay
    global_elapsed = current_time - self.global_last_visit
    if global_elapsed < self.delay:
        time.sleep(self.delay - global_elapsed)

    # Per-URL delay
    if url in self.last_visit_time:
        url_elapsed = current_time - self.last_visit_time[url]
        url_min_delay = self.delay * 3
        if url_elapsed < url_min_delay:
            time.sleep(url_min_delay - url_elapsed)

    # Update timing records
    self.last_visit_time[url] = time.time()
    self.global_last_visit = time.time()

    # Prepare headers with a random user agent
    headers = self.browser_headers.copy()
    headers['User-Agent'] = random.choice(self.user_agents)

    # Add referer for more realistic browsing
    if self.visited_urls:
        # Use a previously visited URL as referer
        potential_referers = list(self.visited_urls)
        if len(potential_referers) > 5:
            potential_referers = potential_referers[-5:]
        referer = random.choice(potential_referers)
        headers['Referer'] = referer

        # Adjust Sec-Fetch-Site based on domain relationship
        referer_domain = urlparse(referer).netloc
        current_domain = urlparse(url).netloc
        if referer_domain == current_domain:
            headers['Sec-Fetch-Site'] = 'same-origin'
        else:
            headers['Sec-Fetch-Site'] = 'cross-site'

    # Create debug directory if enabled and doesn't exist
    debug_dir = "debug_html"
    if not os.path.exists(debug_dir) and len(self.visited_urls) < 10:
        os.makedirs(debug_dir, exist_ok=True)

    try:
        # Add slight variation in the request to appear more natural
        if random.random() < 0.2:
            try:
                self.session.head(
                    url,
                    headers=headers,
                    timeout=random.uniform(1.0, 3.0),
                    allow_redirects=True
                )
                # Small pause between HEAD and GET
                time.sleep(random.uniform(0.1, 0.5))
            except:

                pass

        # Request the page
        response = self.session.get(
            url,
            headers=headers,
            timeout=random.uniform(10.0, 20.0),
            allow_redirects=True
        )

        # Check HTTP status
        if response.status_code != 200:
            logger.warning(f"HTTP error {response.status_code} for {url}")
            self.failed_urls.add(url)
            return "", []

        # Check content type
        content_type = response.headers.get('content-type', '').lower()
        if 'text/html' not in content_type and 'application/xhtml+xml' not in
content_type:
            logger.warning(f"Skipping non-HTML content ({content_type}) at {url}")
            self.failed_urls.add(url)
            return "", []

        # Get HTML content
        html_content = response.text

        # Save raw HTML for debugging
        if len(self.visited_urls) < 5 and debug_dir:
            filename = os.path.join(debug_dir, f"page_{len(self.visited_urls)}.html"
)
            try:
                with open(filename, 'w', encoding='utf-8') as f:
                    f.write(html_content)
                logger.debug(f"Saved debug HTML to {filename}")
            except Exception as debug_e:
                logger.debug(f"Could not save debug HTML: {str(debug_e)}")

        # Extract main content
        clean_text = self._extract_clean_text(html_content, url)

        # Validate content - much less strict validation
        if not self._is_valid_content(clean_text, url):
            logger.warning(f"Invalid content for {url}")
            self.failed_urls.add(url)
            return "", []

        # Extract links
        links = []
```

```python
            try:
                soup = BeautifulSoup(html_content, 'html.parser')

                for a_tag in soup.find_all('a', href=True):
                    href = a_tag['href']

                    # Skip empty or javascript links
                    if not href or href.startswith(('javascript:', '#', 'mailto:', 'tel:')):
                        continue

                    # Convert to absolute URL
                    absolute_url = urljoin(url, href)

                    # Validate URL
                    if self._is_valid_url(absolute_url):
                        # Normalize URL - remove fragments and some query params
                        parsed = urlparse(absolute_url)

                        # Keep only the base URL for uniqueness
                        clean_url = f"{parsed.scheme}://{parsed.netloc}{parsed.path}"

                        # Only add unique URLs
                        if clean_url not in links:
                            links.append(clean_url)
            except Exception as e:
                logger.error(f"Error extracting links from {url}: {str(e)}")

            # Save content to cache
            self.content_cache[url] = clean_text

            # Calculate page quality
            quality = self._get_page_quality_score(clean_text, url)
            self.page_scores[url] = quality

            # Log success
            logger.info(f"Successfully crawled {url} - {len(clean_text)} chars, {len(links)} links, quality: {quality:.2f}")

            return clean_text, links

        except Exception as e:
            logger.error(f"Error fetching {url}: {str(e)}")
            self.failed_urls.add(url)
            return "", []

    def _extract_clean_text(self, html: str, url: str) -> str:
        """Extract clean, main content text from HTML with semantic structure."""
        try:
            # Parse HTML
            soup = BeautifulSoup(html, 'html.parser')

            # Remove non-content elements
            for element in soup.find_all(['script', 'style', 'noscript', 'svg', 'iframe', 'form', 'nav', 'footer']):
                element.decompose()

            # Try to find main content
            main_content = None

            # Check for common content containers - prioritize semantic elements
            for selector in [
                'main', 'article', '#content', '.content', '#main-content', '.main-content',
                '.page-content', '.container', 'div.entry-content', '.article', 'section',
                'body'
            ]:
                elements = soup.select(selector)
                if elements:
                    main_content = elements[0]
                    break

            if not main_content:
                main_content = soup.find(’body’)

            if not main_content:
                logger.warning(f"Could not extract main content from {url}")
                return ""

            # Extract title
            title_text = ""
            title = soup.find(’title’)
            if title and title.text:
                title_text = f"TITLE: {title.text.strip()}\n\n"

            # Get all text with proper spacing
            body_text = main_content.get_text(separator=’\n\n’, strip=True)

            # Combine title and body
            full_text = title_text + body_text

            # Clean up text
            full_text = re.sub(r’\n{3,}’, ’\n\n’, full_text)  # Remove excessive newlines

            return full_text.strip()

        except Exception as e:
            logger.error(f"Error extracting content from {url}: {str(e)}")
            return ""

    def _get_page_quality_score(self, content: str, url: str) -> float:
        """Calculate a quality score for the page (0-1)."""
        if not content:
            return 0.0

        # Base score
        score = 0.5

        # Adjust based on content length (longer is usually better)
        words = len(content.split())
        if words < 100:
            score -= 0.2
        elif words < 300:
            score -= 0.1
        elif words > 500:
            score += 0.1
        elif words > 1000:
            score += 0.2

        # Adjust based on URL priority
        parsed = urlparse(url)
        path = parsed.path.lower()

        # Priority paths get bonus
        if self.priority_paths and any(path.startswith(priority) for priority in self.priority_paths):
            score += 0.15

        # Home page or about page get bonus
        if path == ’/’ or path == ’/index.html’ or path.startswith(’/about’):
            score += 0.1

        # Check for meaningful phrases
        matches = sum(1 for phrase in self.meaningful_phrases if phrase in content.lower())
        phrase_score = min(0.2, matches * 0.02)  # Cap at 0.2
        score += phrase_score

        # Presence of structured content (headings) is usually good
        if "HEADING:" in content:
            score += 0.05

        # Bonus for having a proper title
        if "TITLE:" in content:
```

```python
        score += 0.05

    # Ensure score is in range [0, 1]
    return max(0.0, min(1.0, score))

def _prioritize_urls(self, urls: List[str]) -> List[str]:
    """Prioritize URLs for crawling based on content value heuristics."""
    if not urls:
        return []

    # Score each URL for priority
    scored_urls = []

    for url in urls:
        score = 0.5  # Base score

        # Parse URL
        parsed = urlparse(url)
        path = parsed.path.lower()

        # Prioritize URLs based on path patterns

        # High priority paths
        if any(priority in path for priority in ['/about', '/services', '/resources', '/programs']):
            score += 0.4

        # Medium priority paths
        elif any(priority in path for priority in ['/wellness', '/health', '/student', '/academic']):
            score += 0.3

        # Lower priority but still valuable
        elif any(priority in path for priority in ['/news', '/events', '/contact', '/faq']):
            score += 0.2

        # Deprioritize pagination and archive pages
        if re.search(r'/(page|p)/\d+', path) or re.search(r'/\d{4}/(0\d|1[0-2])', path):
            score -= 0.2

        # Prioritize shorter paths
        path_depth = path.count('/')
        if path_depth <= 1:
            score += 0.1
        elif path_depth >= 4:
            score -= 0.1

        # Check if URL has query parameters
        if parsed.query:
            score -= 0.1

        # Add to scored list
        scored_urls.append((url, score))

    # Sort by score and return URLs only
    sorted_urls = [url for url, score in sorted(scored_urls, key=lambda x: x[1], reverse=True)]

    return sorted_urls

def crawl(self, max_pages: int = 30, max_workers: int = 1):
    """Crawl website with prioritization, quality filters, and improved anti-bot measures."""
    if max_pages <= 0:
        return {}

    # Start with base URL
    urls_to_visit = [self.base_url]

    # Add important paths to initial crawl list
    for path in self.priority_paths:
        if path:
            urls_to_visit.append(urljoin(self.base_url, path))

    # Add common paths that might exist
    common_paths = ["/index.html", "/home", "/about", "/contact", "/resources", "/services"]
    for path in common_paths:
        urls_to_visit.append(urljoin(self.base_url, path))

    # Deduplicate
    urls_to_visit = list(dict.fromkeys(urls_to_visit))

    # Initialize results
    results = {}
    if self.content_cache:
        results = self.content_cache.copy()
        logger.info(f"Starting with {len(results)} cached pages")

    # Check robots.txt first
    self._check_robots_txt()

    # Try alternate domains if base URL has www. or not
    alternate_urls = []
    parsed_base = urlparse(self.base_url)
    base_domain = parsed_base.netloc

    if base_domain.startswith('www.'):
        # Try non-www version
        alt_domain = base_domain[4:]
        alt_url = f"{parsed_base.scheme}://{alt_domain}{parsed_base.path}"
        alternate_urls.append(alt_url)
    else:
        # Try www version
        alt_domain = f"www.{base_domain}"
        alt_url = f"{parsed_base.scheme}://{alt_domain}{parsed_base.path}"
        alternate_urls.append(alt_url)

    # Also try https if the original URL is http
    if parsed_base.scheme == 'http':
        https_url = f"https://{base_domain}{parsed_base.path}"
        alternate_urls.append(https_url)

    # Add alternate URLs to visit list
    for alt_url in alternate_urls:
        if alt_url not in urls_to_visit:
            urls_to_visit.append(alt_url)

    # Crawl sequentially
    with tqdm(total=max_pages, desc="Crawling pages", initial=len(results)) as pbar:
        attempts = 0
        max_attempts = max_pages * 3

        while urls_to_visit and len(results) < max_pages and attempts < max_attempts:
            # First prioritize URLs
            urls_to_visit = self._prioritize_urls(urls_to_visit)

            # Get next URL to visit
            url = urls_to_visit.pop(0)

            # Skip if already visited
            if url in self.visited_urls:
                continue

            # Get page content
            content, links = self._get_page_content(url)
            self.visited_urls.add(url)
            attempts += 1

            if content:
                # Store content
                results[url] = content
                pbar.update(1)
```

```python

                    # Add new links to queue
                    for link in links:
                        if (link not in self.visited_urls and
                            link not in self.failed_urls and
                            link not in urls_to_visit):
                            urls_to_visit.append(link)
                else:
                    self.failed_urls.add(url)

                # Save checkpoint periodically
                if len(results) % 5 == 0:
                    self.content_cache = results
                    self.save_checkpoint()

                # Add a randomized delay between requests
                time.sleep(self._get_random_delay())

        # Update and save final state
        self.content_cache = results
        self.save_checkpoint()

        # Log results
        logger.info(f"Crawling complete: {len(results)} pages, {len(self.visited_urls)} visited, {len(self.failed_urls)} failed")

        # Return results sorted by quality
        sorted_results = {}
        for url, content in sorted(results.items(),
                                key=lambda x: self.page_scores.get(x[0], 0),
                                reverse=True):
            sorted_results[url] = content

        return sorted_results

#----------------------------------------------------------------------
# Persistent Crawler with Checkpointing
#----------------------------------------------------------------------

class PersistentCrawler(EnhancedWebCrawler):
    """Enhanced crawler with checkpoint capabilities."""

    def __init__(self, base_url: str, delay: float = 1.0, checkpoint_dir: str = "checkpoints"):
        """Initialize crawler with checkpoint support."""
        super().__init__(base_url, delay)
        self.checkpoint_dir = checkpoint_dir

        # Create checkpoint directory if it doesn't exist
        os.makedirs(checkpoint_dir, exist_ok=True)

        # Create a unique ID for this crawler instance based on the base URL
        self.checkpoint_id = hashlib.md5(base_url.encode('utf-8')).hexdigest()

    def _get_checkpoint_path(self, file_type: str) -> str:
        """Get the path for a specific checkpoint file."""
        return os.path.join(self.checkpoint_dir, f"{file_type}_{self.checkpoint_id}.json")

    def save_checkpoint(self) -> None:
        """Save crawler state to checkpoint files."""
        try:
            # Save content cache
            cache_path = self._get_checkpoint_path("content")
            with open(cache_path, 'w', encoding='utf-8') as f:
                json.dump(self.content_cache, f)

            # Save visited URLs
            visited_path = self._get_checkpoint_path("visited")
            with open(visited_path, 'w', encoding='utf-8') as f:
                json.dump(list(self.visited_urls), f)

            # Save failed URLs
            failed_path = self._get_checkpoint_path("failed")
            with open(failed_path, 'w', encoding='utf-8') as f:
                json.dump(list(self.failed_urls), f)

            # Save page scores
            scores_path = self._get_checkpoint_path("scores")
            with open(scores_path, 'w', encoding='utf-8') as f:
                json.dump(self.page_scores, f)

            logger.info(f"Checkpoint saved: {len(self.content_cache)} pages")

        except Exception as e:
            logger.error(f"Error saving checkpoint: {str(e)}")

    def load_checkpoint(self) -> bool:
        """Load crawler state from checkpoint files."""
        try:
            # Check if checkpoint files exist
            cache_path = self._get_checkpoint_path("content")
            visited_path = self._get_checkpoint_path("visited")
            failed_path = self._get_checkpoint_path("failed")
            scores_path = self._get_checkpoint_path("scores")

            if not all(os.path.exists(path) for path in [cache_path, visited_path, failed_path]):
                logger.info("Incomplete checkpoint files, starting fresh")
                return False

            # Load content cache
            with open(cache_path, 'r', encoding='utf-8') as f:
                self.content_cache = json.load(f)

            # Load visited URLs
            with open(visited_path, 'r', encoding='utf-8') as f:
                self.visited_urls = set(json.load(f))

            # Load failed URLs
            with open(failed_path, 'r', encoding='utf-8') as f:
                self.failed_urls = set(json.load(f))

            # Load page scores if available
            if os.path.exists(scores_path):
                with open(scores_path, 'r', encoding='utf-8') as f:
                    self.page_scores = json.load(f)

            logger.info(f"Checkpoint loaded: {len(self.content_cache)} pages, {len(self.visited_urls)} visited URLs")
            return True

        except Exception as e:
            logger.error(f"Error loading checkpoint: {str(e)}")
            return False

    def validate_checkpoint(self) -> None:
        """Validate and clean checkpoint data."""
        if not self.content_cache:
            return

        # Check cached content for quality
        invalid_urls = []

        for url, content in tqdm(self.content_cache.items(), desc="Validating checkpoint"):
            if not content or not self._is_valid_content(content, url):
                invalid_urls.append(url)

        # Remove invalid content
        for url in invalid_urls:
            logger.warning(f"Removing invalid content for {url}")
            del self.content_cache[url]
            if url in self.page_scores:
                del self.page_scores[url]

```

```python
        if invalid_urls:
            logger.info(f"Removed {len(invalid_urls)} invalid pages from checkpoint")

            # Save updated checkpoint
            self.save_checkpoint()

#----------------------------------------------------------------------
# Semantic Document Processor
#----------------------------------------------------------------------

class SemanticDocument:
    """A document with semantic understanding for better processing."""

    def __init__(self, text: str, url: str = "", title: str = ""):
        """Initialize a semantic document."""
        self.text = text
        self.url = url
        self.title = title
        self.chunks = []
        self.entities = []
        self.topics = []
        self.embedding = None

    def __str__(self):
        """String representation."""
        return f"Document({self.title or self.url}, {len(self.text)} chars, {len(self.chunks)} chunks)"

    def add_chunk(self, chunk):
        """Add a semantic chunk to the document."""
        self.chunks.append(chunk)

    def add_entity(self, entity):
        """Add an entity to the document."""
        self.entities.append(entity)

    def add_topic(self, topic):
        """Add a topic to the document."""
        self.topics.append(topic)

    def set_embedding(self, embedding):
        """Set the document's embedding vector."""
        self.embedding = embedding

    def get_summary(self):
        """Get a summary of key document statistics."""
        return {
            "url": self.url,
            "title": self.title,
            "length": len(self.text),
            "chunks": len(self.chunks),
            "entities": len(self.entities),
            "topics": [t["text"] for t in self.topics[:5]] if self.topics else []
        }

class SemanticChunk:
    """A semantic chunk of content optimized for QA processing."""

    def __init__(self, text: str, doc_url: str = "", index: int = 0):
        """Initialize a semantic chunk."""
        self.text = text
        self.doc_url = doc_url
        self.index = index
        self.entities = []
        self.embedding = None

    def __str__(self):
        """String representation."""
        return f"Chunk({self.index}, {len(self.text)} chars, {len(self.entities)} entities)"

    def add_entity(self, entity):
        """Add an entity to the chunk."""
        self.entities.append(entity)

    def set_embedding(self, embedding):
        """Set the chunk's embedding vector."""
        self.embedding = embedding

    def to_dict(self):
        """Convert to dictionary for serialization."""
        return {
            "text": self.text,
            "doc_url": self.doc_url,
            "index": self.index,
            "entities": self.entities,
            "embedding": self.embedding.tolist() if isinstance(self.embedding, torch.Tensor) else self.embedding
        }

class DocumentProcessor:
    """Process documents with semantic understanding and chunking."""

    def __init__(self, resource_manager, model_manager: ModelManager = None, use_gpu: bool = True):
        """Initialize document processor with resources."""
        self.resource_manager = resource_manager
        self.model_manager = model_manager or ModelManager(use_gpu=use_gpu)
        self.use_gpu = use_gpu
        self.device = torch.device("cuda" if use_gpu and torch.cuda.is_available() else "cpu")

        # Load embedding model
        self.embedding_model = None
        self.embedding_dimension = 0

        # For tokenization
        self.tokenize_fn = resource_manager.get_fallback_tokenize()

    def load_models(self):
        """Load necessary models for document processing."""
        try:
            # Try to load a more powerful embedding model first
            embedding_model_options = [
                "sentence-transformers/all-mpnet-base-v2",
                "sentence-transformers/all-MiniLM-L12-v2",
                "sentence-transformers/all-MiniLM-L6-v2"
            ]

            for model_name in embedding_model_options:
                try:
                    logger.info(f"Loading embedding model: {model_name}")
                    self.embedding_model = self.model_manager.load_model(
                        "document_embeddings",
                        SentenceTransformer,
                        model_name
                    )

                    # Test the model
                    test_embedding = self.embedding_model.encode("test", convert_to_tensor=True)
                    self.embedding_dimension = test_embedding.shape[0]
                    logger.info(f"Embedding model loaded successfully: {model_name}, dimension: {self.embedding_dimension}")
                    break
                except Exception as e:
                    logger.warning(f"Failed to load embedding model {model_name}: {str(e)}")
                    continue

            if self.embedding_model is None:
                logger.error("Could not load any embedding model")

        except Exception as e:
            logger.error(f"Error loading document processing models: {str(e)}")
```

```python
def process_document(self, text: str, url: str = "", title: str = "") -> SemanticDocument:
    """Process a document into semantic chunks with entity recognition."""
    if not text:
        return None

    # Clean the text first
    text = self._clean_text(text)

    # Initialize document
    doc = SemanticDocument(text, url, title)

    # Extract title if not provided
    if not title:
        title_match = re.search(r'TITLE: (.*?)(\n|$)', text)
        if title_match:
            doc.title = title_match.group(1).strip()

    # Extract entities
    entities = self.resource_manager.analyze_entities(text)
    for entity in entities:
        doc.add_entity(entity)

    # Extract topics/key concepts
    topics = self.resource_manager.extract_key_phrases(text, top_n=10)
    for topic in topics:
        doc.add_topic(topic)

    # Create semantic chunks
    chunks = self._create_semantic_chunks(text, url)
    for chunk in chunks:
        doc.add_chunk(chunk)

    # Create document embedding if model available
    if self.embedding_model:
        try:
            # Use title + first part of text for document-level embedding
            summary_text = (doc.title + ". " if doc.title else "") + text[:1000]
            doc.set_embedding(self.embedding_model.encode(summary_text, convert_to_tensor=True))
        except Exception as e:
            logger.error(f"Error creating document embedding: {str(e)}")

    return doc

def _clean_text(self, text: str) -> str:
    """Clean text for better processing."""
    # Remove excessive whitespace
    text = re.sub(r'\s+', ' ', text)

    # Fix Unicode characters
    text = text.replace('        ', "'").replace('       ', '"').replace('     ', '"')
    text = text.replace('&nbsp;', ' ').replace('&amp;', '&').replace('&quot;', '"')

    # Fix spacing after periods
    text = re.sub(r'\.([A-Z])', r'. \1', text)

    # Handle marked section headers
    text = re.sub(r'HEADING: ', '\n## ', text)
    text = re.sub(r'TITLE: ', '# ', text)
    text = re.sub(r'CONTENT: ', '\n', text)

    return text.strip()

def _create_semantic_chunks(self, text: str, url: str = "") -> List[SemanticChunk]:
    """Create semantic chunks from text preserving context."""
    # This is a key function for improving QA quality through better chunking

    chunks = []

    # Different chunking strategies based on document structure
    if '##' in text or '#' in text:
        # Document has section markers, use them for chunking
        chunks = self._chunk_by_sections(text, url)
    else:
        # Try to identify sections by headings and paragraphs
        chunks = self._chunk_by_paragraphs(text, url)

    # If we get very large chunks, split them further
    max_chunk_size = 1500  # About 300-400 words typically
    new_chunks = []

    for i, chunk in enumerate(chunks):
        if len(chunk.text) > max_chunk_size:
            # Split large chunks with overlap
            sub_chunks = self._split_large_chunk(chunk.text, url, chunk.index)
            new_chunks.extend(sub_chunks)
        else:
            new_chunks.append(chunk)

    # Analyze entities for each chunk
    if new_chunks:
        self._analyze_chunk_entities(new_chunks)

        # Create embeddings for each chunk if model available
        if self.embedding_model:
            try:
                # Prepare all chunks for batch encoding
                texts = [chunk.text for chunk in new_chunks]

                # Encode in batch for efficiency
                embeddings = self.embedding_model.encode(texts, convert_to_tensor=
True)

                # Assign embeddings back to chunks
                for i, chunk in enumerate(new_chunks):
                    chunk.set_embedding(embeddings[i])

            except Exception as e:
                logger.error(f"Error creating chunk embeddings: {str(e)}")

    return new_chunks

def _chunk_by_sections(self, text: str, url: str) -> List[SemanticChunk]:
    """Chunk text by markdown section markers."""
    chunks = []

    # Split by section headers
    section_pattern = r'(^|\n)#+\s+.+?($|\n)'
    sections = re.split(section_pattern, text)

    # Group headers with content
    i = 0
    while i < len(sections):
        if i+2 < len(sections) and re.match(r'(^|\n)#+\s+', sections[i+1]):
            # This is a section header followed by content
            header = sections[i+1].strip()
            content = sections[i+2].strip()

            if content:
                chunk = SemanticChunk(header + "\n" + content, url, len(chunks))
                chunks.append(chunk)

        i += 1

    # If no chunks were created, treat the whole text as one chunk
    if not chunks and text:
        chunks.append(SemanticChunk(text, url, 0))

    return chunks

def _chunk_by_paragraphs(self, text: str, url: str) -> List[SemanticChunk]:
    """Chunk text by paragraphs with some overlap."""
    chunks = []

    # Split into paragraphs first
    paragraphs = []

    # Check if text has natural paragraph breaks
    if ’\n\n’ in text:
        paragraphs = [p.strip() for p in text.split(’\n\n’) if p.strip()]
    else:
        # Try to identify paragraphs by looking for sentence boundaries
        sentences = self.tokenize_fn(text)

        # Group sentences into paragraphs (simple method: ~5 sentences per paragraph)
        current_para = []
        for sentence in sentences:
            current_para.append(sentence)
            if len(current_para) >= 5:
                paragraphs.append(’ ’.join(current_para))
                current_para = []

        # Add the last paragraph if any sentences remain
        if current_para:
            paragraphs.append(’ ’.join(current_para))

    # Group paragraphs into chunks with reasonable sizes
    current_chunk = []
    current_length = 0
    target_length = 1000

    for paragraph in paragraphs:
        para_length = len(paragraph)

        if current_length > 0 and current_length + para_length > target_length:
            chunk_text = ’ ’.join(current_chunk)
            chunks.append(SemanticChunk(chunk_text, url, len(chunks)))

            # Start a new chunk with some overlap
            if current_chunk:
                current_chunk = [current_chunk[-1], paragraph]
                current_length = len(current_chunk[-1]) + para_length
            else:
                current_chunk = [paragraph]
                current_length = para_length
        else:
            # Add paragraph to current chunk
            current_chunk.append(paragraph)
            current_length += para_length

    # Add the last chunk if there’s anything left
    if current_chunk:
        chunk_text = ’ ’.join(current_chunk)
        chunks.append(SemanticChunk(chunk_text, url, len(chunks)))

    return chunks

def _split_large_chunk(self, text: str, url: str, base_index: int) -> List[SemanticChunk]:
    """Split a large chunk into smaller ones with overlapping content."""
    sub_chunks = []

    # Try to split on sentence boundaries
    sentences = self.tokenize_fn(text)

    if not sentences:
        # If tokenization fails, just split by character count
        chunk_size = 1000
        overlap = 100

        for i in range(0, len(text), chunk_size - overlap):
            end = min(i + chunk_size, len(text))
            if end - i < 200:
                break

            sub_text = text[i:end]
```

```python
                idx = base_index * 100 + len(sub_chunks)
                sub_chunks.append(SemanticChunk(sub_text, url, idx))
        else:
            # Split by sentences with overlap
            chunk_size = 10
            overlap = 2

            for i in range(0, len(sentences), chunk_size - overlap):
                end = min(i + chunk_size, len(sentences))
                if end - i < 3:
                    break

                sub_text = ' '.join(sentences[i:end])
                idx = base_index * 100 + len(sub_chunks)
                sub_chunks.append(SemanticChunk(sub_text, url, idx))

        return sub_chunks

    def _analyze_chunk_entities(self, chunks: List[SemanticChunk]) -> None:
        """Extract entities from each chunk."""
        for chunk in chunks:
            entities = self.resource_manager.analyze_entities(chunk.text)
            for entity in entities:
                chunk.add_entity(entity)

#----------------------------------------------------------------------
# Knowledge Base for RAG
#----------------------------------------------------------------------

class KnowledgeBase:
    """Knowledge base for retrieval augmented generation."""

    def __init__(self, use_gpu: bool = True):
        """Initialize knowledge base."""
        self.use_gpu = use_gpu
        self.device = torch.device("cuda" if use_gpu and torch.cuda.is_available() else "cpu")

        # Storage for documents and chunks
        self.documents = {}
        self.chunks = []

        # For vector search
        self.embeddings = None
        self.chunk_ids = []
        self.embedding_dim = 0

        # Text search index
        self.chunk_text_index = {}  # term -> [chunk indices]

    def add_document(self, doc: SemanticDocument) -> None:
        """Add a document to the knowledge base."""
        if not doc or not doc.text:
            return

        # Add document
        self.documents[doc.url] = doc

        # Add chunks
        for chunk in doc.chunks:
            self.chunks.append(chunk)

            # Add to text index
            for term in set(chunk.text.lower().split()):
                if len(term) > 3:
                    if term not in self.chunk_text_index:
                        self.chunk_text_index[term] = []
                    self.chunk_text_index[term].append(len(self.chunks) - 1)

        # Rebuild search index if needed
        self.rebuild_search_index()

    def rebuild_search_index(self) -> None:
        """Rebuild the vector search index."""
        if not self.chunks:
            return

        # Check if chunks have embeddings
        if not hasattr(self.chunks[0], 'embedding') or self.chunks[0].embedding is None:
            logger.warning("Chunks don't have embeddings, can't build search index")
            return

        try:
            # Collect embeddings
            all_embeddings = []
            self.chunk_ids = []

            for i, chunk in enumerate(self.chunks):
                if chunk.embedding is not None:
                    all_embeddings.append(chunk.embedding)
                    self.chunk_ids.append(i)

            if not all_embeddings:
                logger.warning("No valid embeddings found in chunks")
                return

            # Convert to tensor
            if isinstance(all_embeddings[0], list):
                self.embeddings = torch.tensor(all_embeddings)
            else:
                self.embeddings = torch.stack(all_embeddings)

            # Move to device
            if self.use_gpu:
                self.embeddings = self.embeddings.to(self.device)

            # Get embedding dimension
            self.embedding_dim = self.embeddings.shape[1]

            logger.info(f"Built search index with {len(self.chunk_ids)} chunks, dimension: {self.embedding_dim}")

        except Exception as e:
            logger.error(f"Error building search index: {str(e)}")
            self.embeddings = None
            self.chunk_ids = []

    def search(self, query: str, embedding_model=None, top_k: int = 5) -> List[Dict]:
        """Search for relevant chunks using hybrid retrieval."""
        if not self.chunks:
            return []

        # Check if we have embeddings
        if self.embeddings is None:
            # Fallback to text search
            return self._text_search(query, top_k)

        # Check if we have an embedding model
        if embedding_model is None:
            logger.warning("No embedding model provided, falling back to text search")
            return self._text_search(query, top_k)

        try:
            # Get query embedding
            query_embedding = embedding_model.encode(query, convert_to_tensor=True)

            # Move to same device as index
            if self.use_gpu:
                query_embedding = query_embedding.to(self.device)

            # Calculate similarity scores
            similarity = torch.matmul(self.embeddings, query_embedding.unsqueeze(1)).squeeze(1)

            # Get top-k chunks
            if len(similarity) <= top_k:
                top_indices = torch.argsort(similarity, descending=True)
            else:
                top_indices = torch.topk(similarity, k=top_k).indices

            # Convert to Python list
            top_indices = top_indices.cpu().tolist()

            # Get the actual chunk indices
            chunk_indices = [self.chunk_ids[i] for i in top_indices]

            # Get the scores
            scores = [similarity[i].item() for i in top_indices]

            # Create results
            results = []
            for idx, score in zip(chunk_indices, scores):
                chunk = self.chunks[idx]
                results.append({
                    'chunk': chunk,
                    'score': score,
                    'doc_url': chunk.doc_url
                })

            # Hybrid re-ranking: boost scores of chunks matching query terms
            query_terms = set(query.lower().split())
            for result in results:
                # Check if chunk contains query terms
                chunk_text = result['chunk'].text.lower()
                matching_terms = sum(1 for term in query_terms if term in chunk_text)

                # Boost score based on term matches (small boost to preserve vector similarity ordering)
                result['score'] += matching_terms * 0.05

            # Re-sort by adjusted scores
            results.sort(key=lambda x: x['score'], reverse=True)

            return results[:top_k]

        except Exception as e:
            logger.error(f"Error in vector search: {str(e)}")
            # Fallback to text search
            return self._text_search(query, top_k)

    def _text_search(self, query: str, top_k: int = 5) -> List[Dict]:
        """Fallback text-based search."""
        if not self.chunks:
            return []

        # Simple term matching
        query_terms = set(query.lower().split())

        # Score each chunk
        scored_chunks = []

        for i, chunk in enumerate(self.chunks):
            # Count matching terms
            chunk_text = chunk.text.lower()
            matching_terms = sum(1 for term in query_terms if term in chunk_text)

            # Add additional score for exact phrases
            exact_matches = 0
            for size in range(2, min(5, len(query_terms) + 1)):
                for j in range(len(query_terms) - size + 1):
                    phrase = ' '.join(list(query_terms)[j:j+size])
                    if phrase in chunk_text:
                        exact_matches += 1

            # Calculate score
            score = matching_terms + exact_matches * 2

            if score > 0:
                scored_chunks.append((i, score))
```

```python

        # Sort by score
        scored_chunks.sort(key=lambda x: x[1], reverse=True)

        # Get top-k results
        results = []
        for idx, score in scored_chunks[:top_k]:
            chunk = self.chunks[idx]
            results.append({
                'chunk': chunk,
                'score': score,
                'doc_url': chunk.doc_url
            })

        return results

    def get_chunks_by_url(self, url: str) -> List[SemanticChunk]:
        """Get all chunks for a specific URL."""
        return [chunk for chunk in self.chunks if chunk.doc_url == url]

    def get_document(self, url: str) -> Optional[SemanticDocument]:
        """Get a document by URL."""
        return self.documents.get(url)

    def get_stats(self) -> Dict:
        """Get knowledge base statistics."""
        return {
            'documents': len(self.documents),
            'chunks': len(self.chunks),
            'indexed_chunks': len(self.chunk_ids) if self.embeddings is not None else 0,
            'indexed_terms': len(self.chunk_text_index),
            'embedding_dim': self.embedding_dim
        }

    def save(self, filepath: str) -> None:
        """Save knowledge base to file."""
        try:
            # Prepare for serialization
            data = {
                'documents': {},
                'chunks': [],
                'chunk_text_index': self.chunk_text_index
            }

            # Serialize documents (without embeddings to save space)
            for url, doc in self.documents.items():
                data['documents'][url] = {
                    'text': doc.text,
                    'title': doc.title,
                    'topics': doc.topics,
                    'entities': doc.entities
                }

            # Serialize chunks (with embeddings)
            for chunk in self.chunks:
                chunk_data = chunk.to_dict()
                data['chunks'].append(chunk_data)

            # Save to file
            with open(filepath, 'w', encoding='utf-8') as f:
                json.dump(data, f)

            logger.info(f"Knowledge base saved to {filepath}")

        except Exception as e:
            logger.error(f"Error saving knowledge base: {str(e)}")

    def load(self, filepath: str) -> bool:
        """Load knowledge base from file."""
        try:
            # Load from file
            with open(filepath, 'r', encoding='utf-8') as f:
                data = json.load(f)

            # Clear existing data
            self.documents = {}
            self.chunks = []
            self.chunk_text_index = {}
            self.embeddings = None
            self.chunk_ids = []

            # Load documents
            for url, doc_data in data['documents'].items():
                doc = SemanticDocument(doc_data['text'], url, doc_data['title'])
                doc.topics = doc_data['topics']
                doc.entities = doc_data['entities']
                self.documents[url] = doc

            # Load chunks
            for chunk_data in data['chunks']:
                chunk = SemanticChunk(
                    chunk_data['text'],
                    chunk_data['doc_url'],
                    chunk_data['index']
                )
                chunk.entities = chunk_data['entities']

                # Load embedding if present
                if 'embedding' in chunk_data and chunk_data['embedding']:
                    if isinstance(chunk_data['embedding'], list):
                        chunk.embedding = torch.tensor(chunk_data['embedding'])
                    else:
                        # Handle string or other formats
                        logger.warning(f"Unexpected embedding format for chunk {chunk.index}")

                self.chunks.append(chunk)

            # Load chunk text index
            self.chunk_text_index = data.get('chunk_text_index', {})

            # Rebuild search index
            self.rebuild_search_index()

            logger.info(f"Knowledge base loaded from {filepath}: {len(self.documents)} documents, {len(self.chunks)} chunks")
            return True

        except Exception as e:
            logger.error(f"Error loading knowledge base: {str(e)}")
            return False

#-------------------------------------------------------------------
# QA Generator with RAG and CoT
#-------------------------------------------------------------------

class QAGenerator:
    """Advanced QA pair generator using RAG and Chain-of-Thought with enhanced retrieval capabilities."""

    def __init__(self,
                resource_manager,
                model_manager: ModelManager,
                knowledge_base: KnowledgeBase,
                use_gpu: bool = True):
        """Initialize QA generator with necessary components."""
        self.resource_manager = resource_manager
        self.model_manager = model_manager
        self.knowledge_base = knowledge_base
        self.use_gpu = use_gpu
        self.device = torch.device("cuda" if use_gpu and torch.cuda.is_available() else "cpu")

        # Track loaded models
        self.embedding_model = None
        self.question_generator = None
        self.answer_generator = None
        self.qg_tokenizer = None
        self.ag_tokenizer = None

        # For answer generation quality
        self.fact_checker = None
        self.qa_evaluator = None

        # For enhanced retrieval
        self.reranker = None
        self.reranker_tokenizer = None

    def load_models(self):
        """Load necessary models for QA generation with enhanced retrieval capabilities."""
        try:
            # 1. Load embedding model for retrieval
            if self.embedding_model is None:
                # Try to use knowledge base’s existing model first
                if hasattr(self.knowledge_base, ’embedding_model’) and self.knowledge_base.embedding_model:
                    self.embedding_model = self.knowledge_base.embedding_model
                    logger.info("Using knowledge base’s embedding model")
                else:
                    # Otherwise load our own
                    logger.info("Loading embedding model")
                    self.embedding_model = self.model_manager.load_model(
                        "qa_embeddings",
                        SentenceTransformer,
                        "sentence-transformers/all-mpnet-base-v2"
                    )

            # 2. Load question generation model
            if self.question_generator is None:
                logger.info("Loading question generation model")
                try:
                    # Try to load the XXL model first
                    logger.info("Attempting to load Flan-T5-XXL model")
                    self.question_generator = self.model_manager.load_model(
                        "question_generator",
                        T5ForConditionalGeneration,
                        "google/flan-t5-xxl",
                        quantize=True
                    )
                    self.qg_tokenizer = AutoTokenizer.from_pretrained("google/flan-t5-xxl")
                    logger.info("Successfully loaded Flan-T5-XXL model")
                except Exception as e:
                    logger.warning(f"Failed to load T5-XXL model: {e}")
                    try:
                        # Fall back to XL model
                        logger.info("Falling back to Flan-T5-XL model")
                        self.question_generator = self.model_manager.load_model(
                            "question_generator",
                            T5ForConditionalGeneration,
                            "google/flan-t5-xl"
                        )
                        self.qg_tokenizer = AutoTokenizer.from_pretrained("google/flan-t5-xl")
                        logger.info("Successfully loaded Flan-T5-XL model")
                    except Exception as e:
                        logger.warning(f"Failed to load T5-XL model: {e}")
                        # Fall back to an even smaller model
                        try:
                            logger.info("Falling back to Flan-T5-Large model")
                            self.question_generator = self.model_manager.load_model(
                                "question_generator",
                                T5ForConditionalGeneration,
                                "google/flan-t5-large"
                            )
                            self.qg_tokenizer = AutoTokenizer.from_pretrained("google/flan-t5-large")
                            logger.info("Successfully loaded Flan-T5-Large model")
                        except Exception as e:
                            logger.warning(f"Failed to load T5-Large model: {e}")
                            # Final fallback
                            logger.info("Falling back to Flan-T5-Base model")
                            self.question_generator = self.model_manager.load_model(
                                "question_generator",
                                T5ForConditionalGeneration,
                                "google/flan-t5-base"
                            )
                            self.qg_tokenizer = AutoTokenizer.from_pretrained("google/flan-t5-base")
                            logger.info("Successfully loaded Flan-T5-Base model")

            # 3. Load answer generation model (same model can be used for efficiency)
            if self.answer_generator is None:
                logger.info("Loading answer generation model")
                # Re-use the question generator model if it’s suitable
                if self.question_generator and isinstance(self.question_generator, T5ForConditionalGeneration):
                    self.answer_generator = self.question_generator
                    self.ag_tokenizer = self.qg_tokenizer
                    logger.info("Reusing question generation model for answer generation")
                else:
                    # Only executed if question generator failed or is not T5
                    logger.warning("Need to load separate answer generator model")
                    # Try the same cascade of models
                    try:
                        self.answer_generator = self.model_manager.load_model(
                            "answer_generator",
                            T5ForConditionalGeneration,
                            "google/flan-t5-xxl",
                            quantize=True
                        )
                        self.ag_tokenizer = AutoTokenizer.from_pretrained("google/flan-t5-xxl")
                    except Exception as e:
                        logger.warning(f"Failed to load XXL answer generator: {e}")
                        self.answer_generator = self.model_manager.load_model(
                            "answer_generator",
                            T5ForConditionalGeneration,
                            "google/flan-t5-base"
                        )
                        self.ag_tokenizer = AutoTokenizer.from_pretrained("google/flan-t5-base")

            # 4. Optional fact checking model for quality control
            try:
                logger.info("Loading fact checking model")
                self.fact_checker = self.model_manager.load_model(
                    "fact_checker",
                    AutoModelForSequenceClassification,
                    "vectara/hallucination_evaluation_model",
                    trust_remote_code=True
                )
                # Load the tokenizer for this fact-checker
                self.fact_checker_tokenizer = AutoTokenizer.from_pretrained(
                    "vectara/hallucination_evaluation_model",
                    trust_remote_code=True
                )
                logger.info("Successfully loaded fact_checker")
            except Exception as e:
                logger.warning(f"Failed to load fact checker: {e}")
                self.fact_checker = None
                self.fact_checker_tokenizer = None

            # 5. Load cross-encoder re-ranker for enhanced retrieval (higher quality than QA evaluator)
            try:
                logger.info("Loading cross-encoder re-ranker model (L-12)")
                self.reranker = self.model_manager.load_model(
                    "cross_encoder_reranker",
                    CrossEncoder,
                    "cross-encoder/ms-marco-MiniLM-L-12-v2",
                    max_length=512
                )
                logger.info("Successfully loaded cross-encoder re-ranker (L-12)")
            except Exception as e:
                logger.warning(f"Failed to load L-12 cross-encoder re-ranker: {e}")
                # Try a smaller re-ranker model
                try:
                    logger.info("Loading smaller cross-encoder model as fallback")
                    self.reranker = self.model_manager.load_model(
                        "cross_encoder_reranker",
                        CrossEncoder,
                        "cross-encoder/ms-marco-MiniLM-L-6-v2",
                        max_length=512
                    )
                    logger.info("Successfully loaded cross-encoder re-ranker (L-6)")
                except Exception as e:
                    logger.error(f"Failed to load any re-ranker: {e}")
                    self.reranker = None

            # 6. Load QA evaluator for quality assessment (if not already loaded as reranker)
            if self.qa_evaluator is None:
                try:
                    # Directly load MS MARCO as the evaluator
                    self.qa_evaluator = self.model_manager.load_model(
                        "qa_evaluator",
                        AutoModelForSequenceClassification,
                        "cross-encoder/ms-marco-MiniLM-L-6-v2"
                    )
                    self.qa_tokenizer = AutoTokenizer.from_pretrained("cross-encoder/ms-marco-MiniLM-L-6-v2")
                except Exception as e:
                    logger.warning(f"Failed to load QA evaluator: {e}")
                    self.qa_evaluator = None
                    self.qa_tokenizer = None

        except Exception as e:
            logger.error(f"Error loading QA generation models: {str(e)}")
            return False

    def identify_topic_type(self, topic: Dict) -> str:
        """Identify the type of a topic for question generation context."""
        topic_text = topic["text"].lower()

        # Check for explicit type indicators in the text
        type_indicators = {
            "SERVICE": ["service", "center", "office", "desk", "support"],
            "PROGRAM": ["program", "initiative", "project", "series", "system"],
            "RESOURCE": ["resource", "tool", "material", "guide", "handbook"],
            "WELLNESS": ["wellness", "health", "medical", "counseling", "therapy", "wellbeing"],
            "SUPPORT": ["support", "help", "assistance", "aid", "advising"],
            "LOCATION": ["hall", "building", "center", "campus", "laboratory", "library"],
            "CONTACT": ["contact", "email", "phone", "reach", "connect"],
            "ELIGIBILITY": ["eligible", "qualify", "requirement", "criteria"],
            "ACADEMIC": ["academic", "class", "course", "study", "learning", "education"],
            "FINANCIAL": ["financial", "money", "fund", "payment", "cost", "expense", "scholarship"]
        }

        # Check if the topic text contains any type indicators
        for type_name, indicators in type_indicators.items():
            if any(indicator in topic_text for indicator in indicators):
                return type_name

        # If no clear indicators, try to infer from entities
        entity_type_mapping = {
            "ORG": "SERVICE",
            "GPE": "LOCATION",
            "PERSON": "CONTACT",
```

```python
            "DATE": "PROGRAM",
            "MONEY": "FINANCIAL"
        }

        if "entities" in topic and topic["entities"]:
            for entity in topic["entities"]:
                if entity["type"] in entity_type_mapping:
                    return entity_type_mapping[entity["type"]]

        return "GENERAL"

    def generate_questions_from_documents(self, urls: List[str], max_questions_per_url: int = 10) -> List[Dict]:
        """Generate diverse questions from documents."""
        all_questions = []

        for url in urls:
            # Get all chunks for this URL
            chunks = self.knowledge_base.get_chunks_by_url(url)
            if not chunks:
                logger.warning(f"No chunks found for URL: {url}")
                continue

            # Get document if available
            document = self.knowledge_base.get_document(url)

            # Extract document topics if available or analyze chunks
            topics = []
            if document and document.topics:
                topics = document.topics
            else:
                # Analyze chunks to extract topics
                all_text = " ".join([chunk.text for chunk in chunks])
                topics = self.resource_manager.extract_key_phrases(all_text, top_n=15)

            # Generate questions for each important topic
            url_questions = []

            for topic in topics:
                # Identify topic type for context
                topic_type = self.identify_topic_type(topic)

                # Generate questions for this topic
                topic_questions = self.generate_questions_for_topic(
                    topic["text"],
                    topic_type,
                    chunks,
                    max_questions=max(2, max_questions_per_url // len(topics))
                )

                # Add metadata to questions
                for question in topic_questions:
                    question["topic"] = topic["text"]
                    question["topic_type"] = topic_type
                    question["source_url"] = url
                    url_questions.append(question)

            # Generate general questions about the document
            general_questions = self.generate_general_questions(chunks)
            for question in general_questions:
                question["topic"] = "General"
                question["topic_type"] = "GENERAL"
                question["source_url"] = url
                url_questions.append(question)

            # Limit to max questions per URL
            if len(url_questions) > max_questions_per_url:
                # Sort by quality score if available, otherwise keep first ones
                if all("quality_score" in q for q in url_questions):
                    url_questions.sort(key=lambda x: x["quality_score"], reverse=True)
                url_questions = url_questions[:max_questions_per_url]

            all_questions.extend(url_questions)
        return all_questions

    def generate_questions_for_topic(self,
                                     topic: str,
                                     topic_type: str,
                                     chunks: List[SemanticChunk],
                                     max_questions: int = 3) -> List[Dict]:
        """Generate questions for a specific topic using neural models."""
        questions = []

        # Primary approach: Use neural models to generate contextually-appropriate questions
        if self.question_generator and self.qg_tokenizer:
            neural_questions = self._generate_questions_neural(topic, topic_type, chunks)
            questions.extend(neural_questions)

        # If we still don't have enough questions, use rule-based generation as fallback
        if len(questions) < max_questions:
            rule_based_questions = self._generate_questions_rule_based(topic, topic_type, chunks)
            questions.extend(rule_based_questions)

        # Remove duplicates
        unique_questions = self._deduplicate_questions(questions)

        # Ensure all questions end with a question mark
        for q in unique_questions:
            if not q["text"].endswith("?"):
                q["text"] = q["text"] + "?"

        # Return top questions limited by max_questions
        return unique_questions[:max_questions]

    def _generate_questions_neural(self, topic: str, topic_type: str, chunks: List[SemanticChunk]) -> List[Dict]:
        """Generate questions using neural models with enhanced context awareness."""
        if not self.question_generator or not self.qg_tokenizer:
            return []

        questions = []

        try:
            # Extract relevant content about this topic from the chunks
            topic_content = self._extract_topic_content(topic, chunks)
            if not topic_content:
                return []

            # Create a variety of prompts that encourage diverse, natural question generation
            prompts = []

            # General prompt for contextual questions
            prompts.append(
                f"Based on this content about {topic}, generate a natural, informative question that a university student might ask:\n\n"
                f"Content: {topic_content}\n\n"
                f"Question:"
            )

            # Prompt for specific question types based on content analysis
            if topic_type == "SERVICE" or topic_type == "PROGRAM" or topic_type == "RESOURCE":
                prompts.append(
                    f"Create an informative question exploring what {topic} is and how it can benefit students:\n\n"
                    f"Content: {topic_content}\n\n"
                    f"Question:"
                )

                prompts.append(
```

```python
                f"Generate a question about accessing or utilizing {topic} at the university:\n\n"
                f"Content: {topic_content}\n\n"
                f"Question:"
            )

        elif topic_type == "LOCATION":
            prompts.append(
                f"Generate a question about where to find {topic} and what services are available there:\n\n"
                f"Content: {topic_content}\n\n"
                f"Question:"
            )

        elif topic_type == "WELLNESS" or topic_type == "SUPPORT":
            prompts.append(
                f"Create a question asking how {topic} supports student wellbeing or success:\n\n"
                f"Content: {topic_content}\n\n"
                f"Question:"
            )

        elif topic_type == "FINANCIAL":
            prompts.append(
                f"Generate a question about financial aspects of {topic} that would be relevant to students:\n\n"
                f"Content: {topic_content}\n\n"
                f"Question:"
            )

        # Add a chain-of-thought prompt to generate more sophisticated questions
        prompts.append(
            f"Based on this content, create a thoughtful question about {topic}:\n\n"
            f"Content: {topic_content}\n\n"
            f"Step 1: Identify the most important information about {topic}.\n"
            f"Step 2: Consider what students would want to know about {topic}.\n"
            f"Step 3: Formulate a clear, specific question.\n"
            f"Question:"
        )

        # Add a prompt for eligibility or requirements if relevant
        if "eligibility" in topic_content.lower() or "requirement" in topic_content.lower() or "qualify" in topic_content.lower():
            prompts.append(
                f"Create a question about eligibility or requirements for {topic}:\n\n"
                f"Content: {topic_content}\n\n"
                f"Question:"
            )

        # Add a prompt for process-related questions if relevant
        if "process" in topic_content.lower() or "step" in topic_content.lower() or "procedure" in topic_content.lower():
            prompts.append(
                f"Generate a question about the process or steps involved with {topic}:\n\n"
                f"Content: {topic_content}\n\n"
                f"Question:"
            )

        # Generate questions from each prompt
        generated_questions = []

        for prompt in prompts:
            try:
                # Tokenize prompt
                inputs = self.qg_tokenizer(prompt, return_tensors="pt", truncation=True, max_length=1024)
                inputs = {k: v.to(self.device) for k, v in inputs.items()}

                # Generate with diverse sampling parameters
                outputs = self.question_generator.generate(
                    **inputs,
                    max_length=128,
                    num_return_sequences=2,
                    do_sample=True,
                    temperature=0.8,
                    top_p=0.9,
                    no_repeat_ngram_size=3
                )

                # Decode outputs
                for output in outputs:
                    question_text = self.qg_tokenizer.decode(output, skip_special_tokens=True)

                    # Clean up question
                    question_text = self._clean_question(question_text)

                    if question_text:
                        generated_questions.append(question_text)

            except Exception as e:
                logger.error(f"Error generating question from prompt: {str(e)}")
                continue

        # Process and add generated questions
        for question_text in generated_questions:
            questions.append({
                "text": question_text,
                "source": "neural",
                "quality_score": 0.8
            })

        return questions

    except Exception as e:
        logger.error(f"Error in neural question generation: {str(e)}")
        return []

def _extract_topic_content(self, topic: str, chunks: List[SemanticChunk]) -> str:
    """Extract content relevant to a topic from chunks."""
    if not chunks:
        return ""

    # Find chunks that mention the topic
    topic_lower = topic.lower()
    relevant_chunks = []

    for chunk in chunks:
        if topic_lower in chunk.text.lower():
            relevant_chunks.append((chunk, 2))  # Direct mention gets higher score
        else:
            # Check for partial matches (topic terms)
            topic_terms = set(topic_lower.split())
            if len(topic_terms) > 1:  # Only check multi-word topics
                matches = sum(1 for term in topic_terms if term in chunk.text.lower())
                if matches >= len(topic_terms) // 2:  # At least half the terms match
                    relevant_chunks.append((chunk, 1))  # Partial match gets lower score

    # Sort by relevance score
    relevant_chunks.sort(key=lambda x: x[1], reverse=True)

    # Extract text from most relevant chunks (limit length)
    text_parts = []
    total_length = 0
    max_length = 1000

    for chunk, _ in relevant_chunks:
        if total_length + len(chunk.text) > max_length:
            # If adding this chunk would exceed max length, just take enough to reach max
```

```python
                remaining = max_length - total_length
                if remaining > 100:  # Only add if we can get a meaningful amount
                    text_parts.append(chunk.text[:remaining])
                break

            text_parts.append(chunk.text)
            total_length += len(chunk.text)

            if total_length >= max_length:
                break

        # Combine text parts
        return " ".join(text_parts)

    def _generate_questions_rule_based(self, topic: str, topic_type: str, chunks: List[SemanticChunk]) -> List[Dict]:
        """Generate questions using rule-based approaches when neural generation fails."""
        questions = []

        # Extract key sentences from chunks that mention the topic
        topic_lower = topic.lower()
        topic_sentences = []

        tokenize_fn = self.resource_manager.get_fallback_tokenize()

        for chunk in chunks:
            if topic_lower in chunk.text.lower():
                # Get sentences from this chunk
                sentences = tokenize_fn(chunk.text)

                # Find sentences that mention the topic
                for sentence in sentences:
                    if topic_lower in sentence.lower():
                        topic_sentences.append(sentence)

        # Generate questions from key sentences and content patterns
        if topic_sentences:
            for sentence in topic_sentences[:3]:  # Limit to first few sentences
                # Try to identify sentence type and generate appropriate question
                if re.search(r'(is|are|was|were|will be)', sentence.lower()):
                    # Definition/description sentence
                    questions.append({
                        "text": f"What is {topic} and what does it offer?",
                        "source": "rule_based",
                        "quality_score": 0.65
                    })
                elif re.search(r'(can|could|may|might|should)', sentence.lower()):
                    # Capability/possibility sentence
                    questions.append({
                        "text": f"How can students use {topic}?",
                        "source": "rule_based",
                        "quality_score": 0.65
                    })
                elif re.search(r'(located|found|available|offered|provided)', sentence.lower()):
                    # Location/availability sentence
                    questions.append({
                        "text": f"Where can students access {topic}?",
                        "source": "rule_based",
                        "quality_score": 0.65
                    })
                elif re.search(r'(eligible|qualify|qualifies|requirement)', sentence.lower()):
                    # Eligibility sentence
                    questions.append({
                        "text": f"Who is eligible for {topic}?",
                        "source": "rule_based",
                        "quality_score": 0.65
                    })
        else:
            # If no topic-specific sentences found, generate generic questions based on topic type
            if topic_type == "SERVICE":
                questions.append({
                    "text": f"What services does {topic} provide?",
                    "source": "rule_based",
                    "quality_score": 0.6
                })
            elif topic_type == "LOCATION":
                questions.append({
                    "text": f"Where is {topic} located on campus?",
                    "source": "rule_based",
                    "quality_score": 0.6
                })
            elif topic_type == "PROGRAM":
                questions.append({
                    "text": f"What is the purpose of the {topic} program?",
                    "source": "rule_based",
                    "quality_score": 0.6
                })
            else:
                # Generic fallback
                questions.append({
                    "text": f"What information is available about {topic}?",
                    "source": "rule_based",
                    "quality_score": 0.6
                })

        return questions

    def _deduplicate_questions(self, questions: List[Dict]) -> List[Dict]:
        """Remove duplicate and nearly-duplicate questions."""
        if not questions:
            return []

        unique_questions = []
        question_texts = set()

        # First, sort by quality score (highest first)
        questions.sort(key=lambda x: x.get("quality_score", 0), reverse=True)

        for question in questions:
            # Normalize question text
            text = question["text"].lower().strip()

            # Skip if exact duplicate
            if text in question_texts:
                continue

            # Check for near-duplicates
            is_duplicate = False
            for existing in unique_questions:
                if self._questions_are_similar(text, existing["text"].lower()):
                    is_duplicate = True
                    break

            if not is_duplicate:
                question_texts.add(text)
                unique_questions.append(question)

        return unique_questions

    def _questions_are_similar(self, q1: str, q2: str) -> bool:
        """Check if two questions are semantically similar."""
        # Method 1: Jaccard similarity on words
        words1 = set(q1.split())
        words2 = set(q2.split())

        if not words1 or not words2:
            return False

        jaccard = len(words1.intersection(words2)) / len(words1.union(words2))

        # Questions with high word overlap are likely similar
        if jaccard > 0.7:
            return True

        # Method 2: Check edit distance for short questions
        if len(q1) < 50 and len(q2) < 50:
            edit_distance = self._levenshtein_distance(q1, q2)
            if edit_distance / max(len(q1), len(q2)) < 0.3:
                return True

        return False

    def _levenshtein_distance(self, s1: str, s2: str) -> int:
        """Calculate the Levenshtein distance between two strings."""
        if len(s1) < len(s2):
            return self._levenshtein_distance(s2, s1)

        if len(s2) == 0:
            return len(s1)

        previous_row = range(len(s2) + 1)
        for i, c1 in enumerate(s1):
            current_row = [i + 1]
            for j, c2 in enumerate(s2):
                insertions = previous_row[j + 1] + 1
                deletions = current_row[j] + 1
                substitutions = previous_row[j] + (c1 != c2)
                current_row.append(min(insertions, deletions, substitutions))
            previous_row = current_row

        return previous_row[-1]

    def _clean_question(self, question: str) -> str:
        """Clean and normalize a generated question."""
        # Remove any prompt leftovers
        question = re.sub(r'^(Question:|Q:|Step \d+:)', '', question).strip()

        # Ensure first letter is capitalized
        if question and not question[0].isupper():
            question = question[0].upper() + question[1:]

        # Ensure question ends with question mark
        if question and not question.endswith('?'):
            question = question + '?'

        # Remove repetitive question words at start
        question = re.sub(r'^(What|How|Why|Where|When|Who)\s+(is|are|can|does|do|did)\s+\1\s+', r'\1 \2 ', question, flags=re.IGNORECASE)

        return question

    def generate_general_questions(self, chunks: List[SemanticChunk]) -> List[Dict]:
        """Generate general questions about a document using the model."""
        questions = []

        # If we have a neural model, generate contextual general questions
        if self.question_generator and self.qg_tokenizer and chunks:
            try:
                # Combine chunks into a summary
                summary = ""
                total_length = 0
                for chunk in chunks:
                    if total_length > 1500:
                        break
                    summary += chunk.text + " "
                    total_length += len(chunk.text)

                # Create multiple prompts for diverse general questions
                general_prompts = [
                    f"Based on this content, generate a question that would help someone understand the main purpose of this information:\n\nContent: {summary[:1000]}\n\nQuestion:",

                    f"Create a question that would help a student find key resources described in this content:\n\nContent: {summary[500:1500]}\n\nQuestion:",

                    f"Generate a question about how students can access the services mentioned in this content:\n\nContent: {summary[:1000]}\n\nQuestion:",

                    f"Create a question about who students should contact for the services described in this content:\n\nContent: {summary[500:1500]}\n\nQuestion:"
                ]

                # Generate questions from each prompt
                for prompt in general_prompts:
                    inputs = self.qg_tokenizer(prompt, return_tensors="pt", truncation=True, max_length=1024)
                    inputs = {k: v.to(self.device) for k, v in inputs.items()}

                    outputs = self.question_generator.generate(
                        **inputs,
                        max_length=128,
                        num_return_sequences=1,
                        do_sample=True,
                        temperature=0.7
                    )

                    # Process generated question
                    question_text = self.qg_tokenizer.decode(outputs[0], skip_special_tokens=True)
                    question_text = self._clean_question(question_text)

                    if question_text:
                        questions.append({
                            "text": question_text,
                            "source": "neural_general",
                            "quality_score": 0.8
                        })

            except Exception as e:
                logger.error(f"Error generating general neural questions: {str(e)}")
                # Fall back to basic questions if neural generation fails
                questions.append({
                    "text": "What services are described on this page?",
                    "source": "fallback_general",
                    "quality_score": 0.6
                })
                questions.append({
                    "text": "How can students access the resources mentioned here?",
                    "source": "fallback_general",
                    "quality_score": 0.6
                })

        # Deduplicate and return
        return self._deduplicate_questions(questions)[:4]  # Limit to 4 general questions

    def _generate_adaptive_questions(self, urls: List[str]) -> List[Dict]:
        """Generate context-aware adaptive questions for limited content."""
        adaptive_questions = []

        # Get all chunks across all URLs for context
        all_chunks = []
        for url in urls:
            chunks = self.knowledge_base.get_chunks_by_url(url)
            all_chunks.extend(chunks)

        if not all_chunks:
            return []

        # Extract any text content we can find
        all_text = " ".join([chunk.text for chunk in all_chunks])

        # Try neural generation first if model is available
        if self.question_generator and self.qg_tokenizer and all_text:
            try:
                # Extract representative sample of the text
                sample_text = all_text[:2000]  # Take the first 2000 chars as a sample
                # Create adaptive prompts based on available content
                adaptive_prompts = [
                    f"Based on this limited information, generate a general question that would be appropriate regardless of the specific details:\n\nContent: {sample_text}\n\nQuestion:",

                    f"Create a question asking what resources or services are available based on this information:\n\nContent: {sample_text}\n\nQuestion:",

                    f"Generate a question about how to find more information about the topics mentioned here:\n\nContent: {sample_text}\n\nQuestion:"
                ]

                # Look for specific content patterns and create relevant prompts
                if "contact" in all_text.lower() or "email" in all_text.lower() or "phone" in all_text.lower():
                    adaptive_prompts.append(
                        f"Generate a question about how to contact or reach out for the services mentioned:\n\nContent: {sample_text}\n\nQuestion:"
                    )

                if "location" in all_text.lower() or "building" in all_text.lower() or "office" in all_text.lower():
                    adaptive_prompts.append(
                        f"Create a question about where to find the services or offices mentioned:\n\nContent: {sample_text}\n\nQuestion:"
                    )

                # Generate questions from prompts
                for prompt in adaptive_prompts:
                    inputs = self.qg_tokenizer(prompt, return_tensors="pt", truncation=True, max_length=1024)
                    inputs = {k: v.to(self.device) for k, v in inputs.items()}

                    outputs = self.question_generator.generate(
                        **inputs,
                        max_length=128,
                        num_return_sequences=1,
                        do_sample=True,
                        temperature=0.7
                    )

                    question_text = self.qg_tokenizer.decode(outputs[0], skip_special_tokens=True)
                    question_text = self._clean_question(question_text)

                    if question_text:
                        adaptive_questions.append({
                            "text": question_text,
                            "source": "adaptive_neural",
                            "topic": "General",
                            "topic_type": "GENERAL",
                            "quality_score": 0.75,
                            "source_url": urls[0] if urls else ""
                        })

            except Exception as e:
                logger.error(f"Error generating adaptive neural questions: {str(e)}")
                # Fall back to rule-based questions below

        # If we have no questions yet or too few, add some fallback questions
        if len(adaptive_questions) < 3:
            fallback_questions = [
                {
                    "text": "What information is provided on this page?",
                    "source": "adaptive_fallback",
                    "topic": "General",
                    "topic_type": "GENERAL",
                    "quality_score": 0.6,
                    "source_url": urls[0] if urls else ""
                },
                {
                    "text": "What services or resources are described here?",
                    "source": "adaptive_fallback",
                    "topic": "Services",
                    "topic_type": "SERVICE",
                    "quality_score": 0.6,
                    "source_url": urls[0] if urls else ""
                },
                {
                    "text": "How can students get more information about what's mentioned here?",
                    "source": "adaptive_fallback",
                    "topic": "Information",
                    "topic_type": "GENERAL",
                    "quality_score": 0.6,
                    "source_url": urls[0] if urls else ""
                }
            ]

            adaptive_questions.extend(fallback_questions)

        # Extract meaningful terms we can use for additional questions
        keywords = set()
        meaningful_phrases = [
            "student", "university", "campus", "service", "resource", "support",
            "wellness", "health", "academic", "financial", "career", "housing",
            "registration", "advising", "tutoring", "counseling", "aid", "scholarship"
        ]

        for phrase in meaningful_phrases:
            if phrase in all_text.lower():
                keywords.add(phrase)

        # Add keyword-based questions if we found any
        for keyword in list(keywords)[:3]:  # Limit to 3 keywords
            keyword_question = {
                "text": f"What information is provided about {keyword} resources or services?",
                "source": "adaptive_keyword",
                "topic": keyword.title(),
                "topic_type": "KEYWORD",
                "quality_score": 0.65,
                "source_url": urls[0] if urls else ""
            }
            adaptive_questions.append(keyword_question)

        # Deduplicate questions
        unique_questions = self._deduplicate_questions(adaptive_questions)

        return unique_questions

    def generate_answers(self, questions: List[Dict]) -> List[Dict]:
        """Generate answers for a list of questions using RAG."""
        qa_pairs = []

        # Process each question
        for question in tqdm(questions, desc="Generating answers"):
            try:
                # Get question text and metadata
                question_text = question["text"]
                source_url = question.get("source_url", "")

                # 1. Retrieve relevant chunks
                relevant_chunks = self._retrieve_context(question_text, source_url)

                # 2. Generate answer
                answer = self._generate_answer(question_text, relevant_chunks)

                # 3. Evaluate answer quality
                scores = self._evaluate_answer(question_text, answer, relevant_chunks)

                # 4. Create QA pair with metadata
                qa_pair = {
                    "question": question_text,
                    "answer": answer,
                    "source_url": source_url,
                    "topic": question.get("topic", ""),
                    "topic_type": question.get("topic_type", ""),
                    "scores": scores
                }

                qa_pairs.append(qa_pair)

            except Exception as e:
                logger.error(f"Error generating answer for question '{question['text']}': {str(e)}")

        return qa_pairs

    def _fallback_search(self, question: str, source_url: str = "", top_k: int = 10) -> List[Dict]:
        """
        Fallback search method for when the primary search returns too few results.
        Uses more lenient matching and keyword-based approaches.
        """
        fallback_results = []

        # Extract key terms from the question (excluding stopwords)
        question_terms = [term.lower() for term in question.split()
                        if term.lower() not in self.resource_manager.stopwords]

        # Get all chunks
        all_chunks = []
        if source_url:
            # Prioritize chunks from source URL
            source_chunks = self.knowledge_base.get_chunks_by_url(source_url)
            all_chunks.extend([(chunk, 2.0) for chunk in source_chunks])  # Higher weight for source chunks

        # Add other chunks with lower weight
        other_chunks = [chunk for chunk in self.knowledge_base.chunks
                      if not source_url or chunk.doc_url != source_url]
        all_chunks.extend([(chunk, 1.0) for chunk in other_chunks])

        # Score chunks based on term overlap
        scored_chunks = []
        for chunk, base_weight in all_chunks:
            chunk_text = chunk.text.lower()

            # Count matching terms
            matches = sum(1 for term in question_terms if term in chunk_text)
            if matches > 0:
                # Normalize by total terms and apply base weight
                score = (matches / len(question_terms)) * base_weight
                scored_chunks.append({
                    'chunk': chunk,
                    'score': score,
                    'doc_url': chunk.doc_url
                })

        # Sort by score and take top_k
        scored_chunks.sort(key=lambda x: x['score'], reverse=True)
        fallback_results = scored_chunks[:top_k]

        logger.info(f"Fallback search found {len(fallback_results)} additional chunks")
        return fallback_results

    def _retrieve_context(self, question: str, source_url: str = "") -> List[Dict]:
        """
        Context retrieval that searches across the entire knowledge base
        and re-ranks results using a cross-encoder.
        """
        # Initial parameters for retrieval
        initial_top_k = 60
        final_top_k = 30

        # Store retrieved chunks with metadata
```

```python
    retrieved_chunks = []

    # STEP 1: Retrieve chunks from the entire knowledge base
    global_results = self.knowledge_base.search(question, self.embedding_model, top_k=initial_top_k)

    # STEP 2: Process results and mark source-specific chunks
    for result in global_results:
        result['is_source'] = (result['doc_url'] == source_url)
        retrieved_chunks.append(result)

    # STEP 3: If we have very few results, try to increase retrieval scope
    if len(retrieved_chunks) < 3:
        logger.warning(f"Retrieved only {len(retrieved_chunks)} chunks, increasing search scope")
        # Try with more chunks and lower similarity threshold
        additional_results = self._fallback_search(question, source_url, top_k=10)

        # Add non-duplicate chunks
        seen_chunk_ids = {id(chunk['chunk']) for chunk in retrieved_chunks}
        for result in additional_results:
            chunk_id = id(result['chunk'])
            if chunk_id not in seen_chunk_ids:
                result['is_source'] = (result['doc_url'] == source_url)
                retrieved_chunks.append(result)
                seen_chunk_ids.add(chunk_id)

    # STEP 4: Re-rank using cross-encoder if available
    if self.reranker and len(retrieved_chunks) > 1:
        try:
            # Prepare query-passage pairs for re-ranking
            pairs = [(question, chunk['chunk'].text) for chunk in retrieved_chunks]

            # Get cross-encoder scores
            cross_scores = self.reranker.predict(pairs)

            # Update scores with cross-encoder scores
            for i, score in enumerate(cross_scores):
                retrieved_chunks[i]['cross_score'] = float(score)

                # Combine vector similarity with cross-encoder score (weighted)
                vector_score = retrieved_chunks[i]['score']
                if isinstance(vector_score, torch.Tensor):
                    vector_score = vector_score.item()

                # Final score: 0.3 * vector_score + 0.7 * cross_score
                retrieved_chunks[i]['final_score'] = 0.3 * vector_score + 0.7 * float(score)

            # Sort by final score
            retrieved_chunks.sort(key=lambda x: x.get('final_score', 0), reverse=True)

            logger.info(f"Re-ranked {len(retrieved_chunks)} chunks using cross-encoder")

        except Exception as e:
            logger.error(f"Error in cross-encoder re-ranking: {str(e)}")
            # Fall back to original scores
            retrieved_chunks.sort(key=lambda x: x['score'], reverse=True)
    else:
        # If no re-ranker, sort by original scores
        retrieved_chunks.sort(key=lambda x: x['score'], reverse=True)

    # STEP 5: Apply diversity selection to ensure representation from different documents
    diverse_chunks = self._select_diverse_chunks(retrieved_chunks, final_top_k)

    # Log the final number of chunks used for context
    logger.info(f"Using {len(diverse_chunks)} diverse chunks for answering: '{question}'")

    return diverse_chunks
```

```python
def _select_diverse_chunks(self, chunks: List[Dict], max_chunks: int) -> List[Dict]:
    """
    Select a diverse set of chunks using a greedy algorithm that balances
    relevance and diversity across documents.
    """
    if len(chunks) <= max_chunks:
        return chunks

    # Track URLs and chunks already selected
    selected_chunks = []
    selected_urls = set()
    remaining_chunks = chunks.copy()

    # STEP 1: First select the highest scoring chunk overall
    if remaining_chunks:
        best_chunk = max(remaining_chunks, key=lambda x: x.get('final_score', x['score']))
        selected_chunks.append(best_chunk)
        selected_urls.add(best_chunk['doc_url'])
        remaining_chunks.remove(best_chunk)

    # STEP 2: Prioritize source URL chunks if any
    source_chunks = [c for c in remaining_chunks if c.get('is_source', False)]
    if source_chunks:
        # Take the best source chunk
        best_source = max(source_chunks, key=lambda x: x.get('final_score', x['score']))
        if best_source not in selected_chunks:
            selected_chunks.append(best_source)
            remaining_chunks.remove(best_source)

    # STEP 3: Select chunks with a mix of relevance and diversity
    while len(selected_chunks) < max_chunks and remaining_chunks:
        # Calculate diversity bonus for each chunk
        for chunk in remaining_chunks:
            # If this chunk is from a new URL, give it a diversity bonus
            diversity_bonus = 0.3 if chunk['doc_url'] not in selected_urls else 0.0

            # Calculate adjusted score with diversity bonus
            base_score = chunk.get('final_score', chunk['score'])
            chunk['adjusted_score'] = base_score + diversity_bonus

        # Select the chunk with the highest adjusted score
        best_chunk = max(remaining_chunks, key=lambda x: x['adjusted_score'])
        selected_chunks.append(best_chunk)
        selected_urls.add(best_chunk['doc_url'])
        remaining_chunks.remove(best_chunk)

    return selected_chunks

def _is_just_disclaimer(self, text: str) -> bool:
    """Check if the answer is just a disclaimer without actual content."""
    disclaimer_patterns = [
        r"^I don't have (enough|sufficient) information to answer this question\.?\s*$",
        r"^There is not enough (context|information|data) (provided|available|given) to answer this question\.?\s*$",
        r"^Based on the (provided|given|available) (context|information), I cannot answer this question\.?\s*$"
    ]

    # Check if the text matches any disclaimer pattern
    for pattern in disclaimer_patterns:
        if re.match(pattern, text.strip()):
            return True

    # Check length and disclaimer ratio
    words = text.split()
    if len(words) < 20 and "don't have" in text.lower():
        return True

    return False
def _generate_answer(self, question: str, context_chunks: List[Dict]) -> str:
    """Generate an answer using retrieved context with improved prompting."""
    if not context_chunks:
        return "I don’t have enough information to answer this question."

    if not self.answer_generator or not self.ag_tokenizer:
        return "Answer generation model not available."

    try:
        # Get merged context with improved formatting
        context_text = self._merge_context_advanced(question, context_chunks)

        # Prompt with clear instructions
        prompt = (
            f"You are a helpful assistant answering a question based on provided information sources. "
            f"Your task is to synthesize a complete, accurate answer using ONLY the information in the context below. "
            f"Maintain a confident, direct tone and NEVER say ’I don’t have enough information’ if you can provide "
            f"any relevant details from the context.\n\n"

            f"If the information is incomplete, simply share what IS available in the context. "
            f"If the context doesn’t address the question at all, ONLY THEN state that "
            f"you don’t have the specific information requested.\n\n"

            f"CONTEXT:\n{context_text}\n\n"

            f"INSTRUCTIONS FOR ANSWERING:\n"
            f"1. Read the context carefully and identify all relevant information\n"
            f"2. Synthesize the information into a coherent, complete answer\n"
            f"3. If information is partial, provide what’s available without disclaimers\n"
            f"4. Ensure your answer is fully supported by the context\n"
            f"5. Write in complete sentences with proper formatting\n\n"

            f"QUESTION: {question}\n\n"
            f"ANSWER:"
        )

        # Tokenize prompt with increased max length to handle larger context
        inputs = self.ag_tokenizer(prompt, return_tensors="pt", truncation=True, max_length=3072)
        inputs = {k: v.to(self.device) for k, v in inputs.items()}


        outputs = self.answer_generator.generate(
            **inputs,
            max_length=1000,
            min_length=50,
            num_beams=4,
            num_beam_groups=1,
            num_return_sequences=2,
            diversity_penalty=0.0,
            do_sample=False,
            temperature=1.0,
            top_p=1.0,
            no_repeat_ngram_size=3,
            length_penalty=1.5,
            early_stopping=True
        )

        # Process candidates
        candidates = []
        for output in outputs:
            answer_text = self.ag_tokenizer.decode(output, skip_special_tokens=True)
            candidates.append(answer_text)

        # Select the best answer
        if candidates:
            # Filter out candidates that are just disclaimers
            valid_candidates = [c for c in candidates if not self._is_just_disclaimer(c)]

            if valid_candidates:
                # Choose the most substantive answer
                best_answer = max(valid_candidates, key=lambda x: len(x) - 10 * x.count("I don't have"))
            else:
                best_answer = candidates[0]  # Fallback to first answer

            # Apply formatting
            return self._format_answer(best_answer)
        else:
            return "I couldn't generate an answer based on the available information."

    except Exception as e:
        logger.error(f"Error generating answer: {str(e)}")
        return "I encountered an error while generating the answer."

def _clean_chunk_text(self, text: str) -> str:
    """Clean chunk text of artifacts and normalize formatting."""
    # Remove HTML artifacts
    text = re.sub(r'&nbsp;|&amp;|&lt;|&gt;|&quot;', ' ', text)

    # Fix ellipsis and other punctuation
    text = re.sub(r'\.{2,}', '. ', text)

    # Normalize whitespace
    text = re.sub(r'\s+', ' ', text)

    # Fix newlines
    text = re.sub(r'\r\n|\r', '\n', text)
    text = re.sub(r'\n{3,}', '\n\n', text)

    # Remove artifacts common in the data
    text = re.sub(r'rn\s', ' ', text)
    text = re.sub(r'\.s\.', '.', text)

    # Fix broken markdown headers
    text = re.sub(r'#\s+', '## ', text)

    return text.strip()

def _is_text_too_similar(self, text1: str, text2: str) -> bool:
    """Check if two texts are too similar to both include."""
    # For very short texts, use exact matching
    if len(text1) < 100 or len(text2) < 100:
        return text1 in text2 or text2 in text1

    # For longer texts, use n-gram similarity
    words1 = text1.split()
    words2 = text2.split()

    # Create 3-grams
    def get_ngrams(words, n=3):
        return set(' '.join(words[i:i+n]) for i in range(len(words)-n+1))

    ngrams1 = get_ngrams(words1)
    ngrams2 = get_ngrams(words2)

    if not ngrams1 or not ngrams2:
        return False

    # Calculate Jaccard similarity
    intersection = len(ngrams1.intersection(ngrams2))
    union = len(ngrams1.union(ngrams2))

    # Higher threshold to avoid removing related but distinct content
    return intersection / union > 0.8

def _merge_context_advanced(self, question: str, context_chunks: List[Dict]) -> str:
```

```python
    """Advanced context merging with clear source boundaries and improved structure."""
    if not context_chunks:
        return ""

    # Sort chunks by relevance
    sorted_chunks = sorted(context_chunks, key=lambda x: x.get('final_score', x['score']), reverse=True)

    # Calculate maximum context size based on model
    if self.answer_generator and hasattr(self.answer_generator, 'config'):
        if hasattr(self.answer_generator.config, 'model_type'):
            model_path = getattr(self.answer_generator.config, '_name_or_path', '').lower()
            if 'xxl' in model_path:
                max_context_chars = 30000
                logger.info("Using expanded context size for XXL model")
            else:
                max_context_chars = 18000
        else:
            max_context_chars = 18000
    else:
        max_context_chars = 18000

    # Group chunks by source document
    doc_chunks = {}
    for chunk_data in sorted_chunks:
        chunk = chunk_data['chunk']
        doc_url = chunk.doc_url
        if doc_url not in doc_chunks:
            doc_chunks[doc_url] = []
        doc_chunks[doc_url].append((chunk, chunk_data.get('final_score', chunk_data['score'])))

    # Assemble context with clear document sections
    context_parts = []
    current_length = 0

    # First add chunks from the highest scoring documents
    for doc_url, chunks in sorted(doc_chunks.items(),
                                key=lambda x: max([score for _, score in x[1]]),
                                reverse=True):
        # Sort chunks within this document by score
        doc_chunks_sorted = sorted(chunks, key=lambda x: x[1], reverse=True)

        # Extract domain for reference
        domain = doc_url.replace('https://', '').replace('http://', '').split('/')[0]
        path = doc_url.split('/')[-1] if '/' in doc_url else ''

        # Clean and combine text from this document
        doc_texts = []
        for chunk, _ in doc_chunks_sorted:
            # Clean and normalize text
            text = self._clean_chunk_text(chunk.text)

            # Check if this would exceed our max context
            if current_length + len(text) + 100 > max_context_chars:
                # If we already have content, just stop adding more
                if doc_texts or context_parts:
                    break
                # If this is the first chunk, take a portion to fit
                truncated = text[:max_context_chars - 200] + "..."
                doc_texts.append(truncated)
                current_length += len(truncated)
                break

            # Add text if not too similar to existing content
            if not any(self._is_text_too_similar(text, existing) for existing in doc_texts):
                doc_texts.append(text)
                current_length += len(text)

        # Only add this document section if we have content
        if doc_texts:
            # Create section header
            section_header = f"DOCUMENT: {domain}/{path}"
            section_content = "\n\n".join(doc_texts)
            section = f"{'='*50}\n{section_header}\n{'='*50}\n{section_content}"
            context_parts.append(section)

    # Join all sections with clear separation
    full_context = "\n\n" + "\n\n".join(context_parts)

    # Add helpful metadata at the beginning
    context_intro = (
        f"QUESTION: {question}\n\n"
        f"The following information comes from {len(doc_chunks)} different sources about this topic. "
        f"Use this information to construct a complete, accurate answer."
    )

    full_context = context_intro + full_context

    return full_context

def _split_into_semantic_units(self, text: str) -> List[str]:
    """Split text into semantic units (paragraphs or coherent sections)."""
    # First try to split by paragraph breaks
    if '\n\n' in text:
        segments = [seg.strip() for seg in text.split('\n\n') if seg.strip()]
        # Filter out very short segments and merge them with adjacent ones
        filtered_segments = []
        current_segment = ""

        for segment in segments:
            if len(segment) < 50:  # Short segment
                current_segment += " " + segment
            else:
                if current_segment:
                    filtered_segments.append(current_segment)
                    current_segment = segment
                else:
                    current_segment = segment

        # Add the last segment if it exists
        if current_segment:
            filtered_segments.append(current_segment)

        return filtered_segments if filtered_segments else [text]

    # If no paragraph breaks, try to use sentence tokenization
    tokenize_fn = self.resource_manager.get_fallback_tokenize()
    sentences = tokenize_fn(text)

    if len(sentences) <= 3:
        # Text is already small enough
        return [text]

    # Group sentences into coherent segments
    segments = []
    current_segment = []
    for sentence in sentences:
        current_segment.append(sentence)
        if len(current_segment) >= 4:
            segments.append(" ".join(current_segment))
            current_segment = []

    # Add the last segment if it exists
    if current_segment:
        segments.append(" ".join(current_segment))

    return segments if segments else [text]

def _eliminate_redundancy(self, blocks: List[Dict]) -> List[Dict]:
    """Detect and eliminate redundant content from blocks."""
    if len(blocks) <= 1:
        return blocks

    # Sort blocks by score (highest first)
    sorted_blocks = sorted(blocks, key=lambda x: x['score'], reverse=True)

    # Calculate similarity threshold
    similarity_threshold = 0.7

    # Track which blocks are redundant
    is_redundant = [False] * len(sorted_blocks)

    # For each high-scoring block, check if lower-scoring blocks are redundant
    for i in range(len(sorted_blocks) - 1):
        if is_redundant[i]:
            continue

        block_i = sorted_blocks[i]['text'].lower()

        for j in range(i + 1, len(sorted_blocks)):
            if is_redundant[j]:
                continue

            block_j = sorted_blocks[j]['text'].lower()

            # Simple n-gram based similarity check
            similarity = self._calculate_text_similarity(block_i, block_j)

            # Mark as redundant if similarity is high
            if similarity > similarity_threshold:
                # If blocks are from same chunk, always mark the lower-scoring one
                if sorted_blocks[i]['chunk_id'] == sorted_blocks[j]['chunk_id']:
                    is_redundant[j] = True
                else:
                    score_ratio = sorted_blocks[j]['score'] / sorted_blocks[i]['score']
                    if score_ratio > 0.85 and len(block_j) < len(block_i) * 0.8:
                        # Keep the shorter block if scores are comparable
                        continue
                    else:
                        is_redundant[j] = True

    # Return non-redundant blocks
    return [block for i, block in enumerate(sorted_blocks) if not is_redundant[i]]

def _calculate_text_similarity(self, text1: str, text2: str) -> float:
    """Calculate similarity between two text blocks using n-grams."""
    # Use tf-idf weighted bigram similarity for a balance of efficiency and accuracy

    # Get bigrams
    def get_bigrams(text):
        words = text.split()
        return set(" ".join(words[i:i+2]) for i in range(len(words)-1))

    bigrams1 = get_bigrams(text1)
    bigrams2 = get_bigrams(text2)

    # Handle empty sets
    if not bigrams1 or not bigrams2:
        return 0.0

    # Calculate Jaccard similarity
    intersection = len(bigrams1.intersection(bigrams2))
    union = len(bigrams1.union(bigrams2))

    return intersection / union

def _select_context_blocks(self, question: str, blocks: List[Dict]) -> List[Dict]:
    """
    Select context blocks to include, balancing relevance and diversity
    while managing context length.
    """
    if self.answer_generator and hasattr(self.answer_generator, 'config'):
        if hasattr(self.answer_generator.config, 'model_type'):
            model_path = getattr(self.answer_generator.config, '_name_or_path', '').lower()
            if 'xxl' in model_path:
                max_context_length = 3000
            elif 'xl' in model_path:
                max_context_length = 2000
            elif 'large' in model_path:
                max_context_length = 1500
            else:
                max_context_length = 1000
        else:
            max_context_length = 1500
    else:
        max_context_length = 1500

    # If we have few blocks, include them all if they fit
    total_length = sum(len(block['text']) for block in blocks)
    if total_length <= max_context_length:
        return blocks

    # We need to be selective - first prioritize blocks from different documents
    doc_urls = set(block['doc_url'] for block in blocks)

    selected_blocks = []
    remaining_blocks = blocks.copy()
    current_length = 0

    # First phase: Select highest scoring block from each document
    for url in doc_urls:
        doc_blocks = [b for b in remaining_blocks if b['doc_url'] == url]
        if doc_blocks:
            best_block = max(doc_blocks, key=lambda x: x['score'])
            if current_length + len(best_block['text']) <= max_context_length:
                selected_blocks.append(best_block)
                remaining_blocks.remove(best_block)
                current_length += len(best_block['text'])

    # Second phase: Select additional blocks based on score, with diminishing returns
    # for blocks from the same document
    remaining_blocks.sort(key=lambda x: x['score'], reverse=True)

    # Count blocks per document
    doc_counts = Counter(block['doc_url'] for block in selected_blocks)

    # Adjust scores based on document representation
    for block in remaining_blocks:
        # Apply penalty for documents that are already well-represented
        doc_count = doc_counts.get(block['doc_url'], 0)
        adjusted_score = block['score'] * (0.95 ** doc_count)  # Diminishing returns
        block['adjusted_score'] = adjusted_score

    # Re-sort with adjusted scores
    remaining_blocks.sort(key=lambda x: x.get('adjusted_score', x['score']), reverse=True)

    # Add blocks until we reach the length limit
    for block in remaining_blocks:
        if current_length + len(block['text']) <= max_context_length:
            selected_blocks.append(block)
            current_length += len(block['text'])
            # Update document count
            doc_counts[block['doc_url']] = doc_counts.get(block['doc_url'], 0) + 1
        else:
            # Try to fit as much relevant content as possible
            space_left = max_context_length - current_length
            if space_left > 200:  # Only add if we can fit something substantial
                # Truncate the block to fit
                truncated_text = block['text'][:space_left].rsplit('.', 1)[0] + '.'
                if len(truncated_text) > 100:  # Only add if it's still meaningful
                    block['text'] = truncated_text
                    selected_blocks.append(block)
```

```python
                break

        return selected_blocks

    def _assemble_context(self, blocks: List[Dict]) -> str:
        """Assemble the final context, organized by document source."""
        if not blocks:
            return ""

        # Group blocks by document URL
        blocks_by_url = {}
        for block in blocks:
            url = block['doc_url']
            if url not in blocks_by_url:
                blocks_by_url[url] = []
            blocks_by_url[url].append(block)

        # Assemble context with source information
        context_parts = []

        for url, url_blocks in blocks_by_url.items():
            # Sort blocks from same document by their original order
            url_blocks.sort(key=lambda x: (x['chunk_id'], x['segment_id']))

            # Combine text from this document
            doc_text = " ".join(block['text'] for block in url_blocks)

            # Add document source indicator for multi-document context
            if len(blocks_by_url) > 1:
                # Extract domain for cleaner reference
                domain = url.replace('https://', '').replace('http://', '').split('/')[0]
                source_indicator = f"[From: {domain}] "
                context_parts.append(source_indicator + doc_text)
            else:
                context_parts.append(doc_text)

        # Join all parts
        return "\n\n".join(context_parts)

    def _prepare_context(self, question: str, chunks: List[Dict]) -> str:
        """
        Legacy method kept for compatibility.
        Now calls the advanced context merging function.
        """
        return self._merge_context_advanced(question, chunks)

    def _extract_answer_from_cot(self, text: str) -> str:
        """Extract the final answer from a chain-of-thought generation."""
        # If text contains step markers, extract the part after the last step
        step_matches = list(re.finditer(r'Step \d+:', text))
        if step_matches:
            last_step_match = step_matches[-1]
            last_step_end = last_step_match.end()

            # Find the next step or the end of text
            next_step_match = re.search(r'Step \d+:', text[last_step_end:])
            if next_step_match:
                # Extract between last step and next step
                answer_text = text[last_step_end:last_step_end + next_step_match.start()].strip()
            else:
                # Extract from last step to end
                answer_text = text[last_step_end:].strip()

            return answer_text

        # If we see a clear "Answer:" marker
        answer_match = re.search(r'Answer:(.*?)(?:$|Step \d+:)', text, re.DOTALL)
        if answer_match:
            return answer_match.group(1).strip()

        # If none of the above patterns match, just return the original text
        return text

    def _select_best_answer(self, question: str, candidates: List[str], context: str) -> str:
        """Select the best answer from multiple candidates based on quality and consistency."""
        if not candidates:
            return ""

        if len(candidates) == 1:
            return candidates[0]

        # 1. Score candidates by length (prefer longer, more detailed answers)
        length_scores = []
        for candidate in candidates:
            # Normalize length (prefer answers between 50-200 characters)
            length = len(candidate)
            if length < 20:
                score = length / 20  # Penalize very short answers
            elif length < 50:
                score = 0.5 + (length - 20) / 60  # Ramp up to 0.5-1.0
            elif length <= 200:
                score = 1.0  # Ideal length
            else:
                score = 1.0 - (length - 200) / 800  # Gradually penalize very long answers
                score = max(0.5, score)  # Don't go below 0.5

            length_scores.append(score)

        # 2. Check factual consistency with context
        factual_scores = []
        for candidate in candidates:
            # Simple heuristic: count overlapping n-grams with context
            candidate_ngrams = self._get_ngrams(candidate.lower(), 2)
            context_ngrams = self._get_ngrams(context.lower(), 2)

            overlap = len(candidate_ngrams.intersection(context_ngrams))
            total = len(candidate_ngrams)

            if total == 0:
                factual_scores.append(0.0)
            else:
                factual_scores.append(min(1.0, overlap / total))

        # 3. Combined scoring
        final_scores = []
        for i in range(len(candidates)):
            # Weight factual consistency more heavily
            score = 0.3 * length_scores[i] + 0.7 * factual_scores[i]
            final_scores.append(score)

        # Return the candidate with the highest score
        best_idx = final_scores.index(max(final_scores))
        return candidates[best_idx]

    def _get_ngrams(self, text: str, n: int) -> set:
        """Get n-grams from text."""
        words = text.split()
        ngrams = set()
        for i in range(len(words) - n + 1):
            ngram = ' '.join(words[i:i+n])
            ngrams.add(ngram)
        return ngrams

    def _fix_structural_issues(self, text: str) -> str:
        """Fix structural issues in the answer."""
        # Remove any instruction artifacts
        text = re.sub(r'^(Answer:|ANSWER:|Step \d+:|To answer this question:)', '', text).strip()

        # Remove meta-commentary about the answering process
        text = re.sub(r’(Based on the (provided|given|available) (context|information),?\s*)’, ’’, text).strip()
        text = re.sub(r’(According to the (provided|given|available) (context|information),?\s*)’, ’’, text).strip()

        # Remove duplicate phrases at the beginning
        words = text.split()
        if len(words) > 10:
            first_5 = ’ ’.join(words[:5]).lower()
            for i in range(1, min(10, len(words) - 5)):
                next_5 = ’ ’.join(words[i:i+5]).lower()
                if first_5 == next_5:
                    text = ’ ’.join(words[i:])
                    break

        return text

    def _clean_formatting_artifacts(self, text: str) -> str:
        """Clean formatting artifacts from the answer."""
        # Fix newlines and spacing
        text = re.sub(r’\r\n|\r’, ’\n’, text)
        text = re.sub(r’\n{2,}’, ’\n\n’, text)
        text = re.sub(r’[ \t]+’, ’ ’, text)

        # Fix ellipses and quotes
        text = re.sub(r’\.{2,}’, ’. ’, text)
        text = re.sub(r’"{2,}’, ’"’, text)
        text = re.sub(r"’{2,}", "’", text)

        # Fix common artifacts
        text = re.sub(r’\.s\.’, ’.’, text)
        text = re.sub(r’\bn\s’, ’ ’, text)
        text = re.sub(r’rn’, ’’, text)

        # Fix list formatting
        text = re.sub(r’(\d+)\)\s*’, r’\1. ’, text)

        return text.strip()

    def _fix_truncated_ending(self, text: str) -> str:
        """Fix truncated endings to ensure complete sentences."""
        if not text:
            return text

        # If text doesn’t end with sentence-ending punctuation
        if not re.search(r’[.!?]$’, text):
            # Try to find the last complete sentence
            last_period = max(
                text.rfind(’.’),
                text.rfind(’!’),
                text.rfind(’?’)
            )

            if last_period > 0.7 * len(text):
                # If we have most of the content, trim to last complete sentence
                return text[:last_period+1]
            else:
                # Otherwise add a period to the existing text
                return text + "."

        return text

    def _fix_capitalization_punctuation(self, text: str) -> str:
        """Fix capitalization and punctuation in the answer."""
        # Capitalize first letter
        if text and text[0].isalpha() and not text[0].isupper():
            text = text[0].upper() + text[1:]

        # Remove filler conclusions
        conclusion_patterns = [
            r’ I hope (this|that) (helps|answers your question)\.?$’,
            r’ (Please )?[Ll]et me know if you (have|need) (any|more|further|additional) (questions|information|details)\.?$’,
            r' (Feel free to|Please) (ask|contact|reach out)( me)?( if you have| for)? (more|any other|additional) questions\.?$'
        ]

        for pattern in conclusion_patterns:
            text = re.sub(pattern, '.', text)

        return text

    def _format_answer(self, answer: str) -> str:
        """Format the answer with improved handling of disclaimers and artifacts."""
        if not answer:
            return "I don't have specific information about this in the available sources."

        # Extract meaningful content after disclaimers
        if answer.startswith("I don't have enough information"):
            # Look for content after the disclaimer
            parts = answer.split(".", 1)
            if len(parts) > 1 and len(parts[1].strip()) > 30:
                # If substantial content follows the disclaimer, use it
                answer = parts[1].strip()
            else:
                # If there's nothing substantial, keep the disclaimer but improve it
                return "Based on the available sources, I don't have specific information to answer this question completely."

        # Fix answer structure issues
        answer = self._fix_structural_issues(answer)

        # Clean up formatting artifacts
        answer = self._clean_formatting_artifacts(answer)

        # Ensure the answer doesn't end mid-sentence
        answer = self._fix_truncated_ending(answer)

        # Ensure proper capitalization and punctuation
        answer = self._fix_capitalization_punctuation(answer)

        return answer

    def _evaluate_answer(self, question: str, answer: str, context_chunks: List[Dict]) -> Dict[str, float]:
        """Evaluate answer quality with enhanced checks for formatting issues and completeness."""
        # Convert context_chunks format for compatibility with the original method
        semantic_chunks = [chunk_data['chunk'] for chunk_data in context_chunks]

        scores = {
            'relevance': 0.0,
            'factuality': 0.0,
            'completeness': 0.0,
            'formatting': 0.0,
            'overall': 0.0
        }

        # Check for empty or very short answers
        if not answer or len(answer) < 15:
            return scores

        # Check if we have limited context - adjust scoring if needed
        limited_context = len(context_chunks) <= 1

        # 1. Relevance score - more lenient with limited content
        relevance = self._score_relevance(question, answer)
        if limited_context:
            # Boost relevance score for limited content
            relevance = min(1.0, relevance * 1.2)
        scores['relevance'] = relevance

        # 2. Factuality score - adjust for limited content
        factuality = self._score_factuality(answer, semantic_chunks)
        if limited_context:
            # Set a minimum factuality score for limited content
            factuality = max(0.5, factuality)
        scores['factuality'] = factuality

        # 3. Completeness score - with enhanced checks for truncation
        completeness = self._score_completeness(question, answer)

        # Check for truncated answers or formatting issues
        formatting_score = 1.0

        # Penalize answers that appear truncated
        if len(answer) > 20 and not answer.endswith(('.', '!', '?')):
            completeness *= 0.8
            formatting_score *= 0.7

        # Penalize answers with "I don't have enough information" followed by content
        if "I don't have enough information" in answer and len(answer) > 70:
            completeness *= 0.9
            formatting_score *= 0.8

        # Penalize answers with strange formatting artifacts
        formatting_artifacts = [
            r'\.{3,}',
            r'\brn\b',
            r'\.s\.',
            r'\n{3,}',
            r'\s{3,}'
        ]

        for pattern in formatting_artifacts:
            if re.search(pattern, answer):
                formatting_score *= 0.85

        if limited_context:
            # Boost completeness for limited content
            completeness = min(1.0, completeness * 1.2)

        scores['completeness'] = completeness
        scores['formatting'] = formatting_score

        # 4. Calculate overall score with weighted combination
        if limited_context:
            # For limited content, weigh relevance more heavily
            overall = (0.35 * relevance + 0.25 * factuality + 0.25 * completeness + 0.15 * formatting_score)
        else:
            # Standard weighting
            overall = (0.25 * relevance + 0.35 * factuality + 0.25 * completeness + 0.15 * formatting_score)

        scores['overall'] = overall

        return scores

    def _score_relevance(self, question: str, answer: str) -> float:
        """Score the relevance of an answer to the question."""
        # Method 1: Check for question terms in the answer
        question_terms = set(question.lower().split()) - self.resource_manager.stopwords
        answer_lower = answer.lower()

        # Count matching terms
        matching_terms = sum(1 for term in question_terms if term in answer_lower)
        term_score = min(1.0, matching_terms / max(1, len(question_terms)))

        # Method 2: Use semantic model if available
        if self.qa_evaluator:
            try:
                # Use the model to score the pair
                inputs = self.qa_tokenizer([question], [answer], return_tensors="pt", padding=True, truncation=True)
                inputs = {k: v.to(self.device) for k, v in inputs.items()}

                outputs = self.qa_evaluator(**inputs)
```

```python
            logits = outputs.logits

            # Convert logits to probability
            if logits.shape[1] > 1:
                # Multi-class classification
                probs = torch.softmax(logits, dim=1)
                model_score = probs[0, 1].item()  # Assume second class is "relevant"
            else:
                # Binary classification
                prob = torch.sigmoid(logits)
                model_score = prob.item()

            # Combine model score with term score
            return 0.7 * model_score + 0.3 * term_score

        except Exception as e:
            logger.error(f"Error using QA evaluator: {str(e)}")
            return term_score

    return term_score

def _score_factuality(self, answer: str, context_chunks: List[SemanticChunk]) -> float:
    """Score the factual consistency of the answer with the context."""
    if not context_chunks:
        return 0.5  # Neutral if no context

    # Combine context
    context = " ".join(chunk.text for chunk in context_chunks)

    # Use fact checking model if available
    if self.fact_checker and self.fact_checker_tokenizer:
        try:
            # Merge claim + partial context into a single string
            combined_text = f"Claim: {answer}\nContext: {context[:1000]}"
            inputs = self.fact_checker_tokenizer(
                combined_text, return_tensors="pt", truncation=True, max_length=512
            )
            inputs = {k: v.to(self.device) for k, v in inputs.items()}
            outputs = self.fact_checker(**inputs)

            # Check outputs.logits
            if hasattr(outputs, "logits"):
                logits = outputs.logits
                # If single-logit, interpret > 0 => more truthful
                if logits.shape[-1] == 1:
                    prob = torch.sigmoid(logits).item()
                    return prob
                else:
                    # Multi-logit => assume index 1 is "not hallucinated" or "truthful"
                    probs = torch.softmax(logits, dim=-1)
                    return probs[0, 1].item()

            return 0.5

        except Exception as e:
            logger.error(f"Error using fact checker: {str(e)}")
            # Fall back to n-gram overlap below

    # If fact checker not available or it failed
    answer_ngrams = self._get_ngrams(answer.lower(), 2)
    context_ngrams = self._get_ngrams(context.lower(), 2)
    if not answer_ngrams:
        return 0.0

    overlap = len(answer_ngrams.intersection(context_ngrams))
    score = overlap / len(answer_ngrams)
    return 0.2 + (score * 0.8)


def _score_completeness(self, question: str, answer: str) -> float:
    """
    Score how completely the answer addresses all aspects of the question
    with enhanced detection of truncated or malformed answers.
    """
    # Base score
    score = 0.5

    # 1. Length-based scoring (very short answers are likely incomplete)
    words = len(answer.split())
    if words < 15:
        score -= 0.3
    elif words > 50:
        score += 0.1

    # 2. Check for truncated answers
    if not answer.endswith((’.’, ’!’, ’?’)) and len(answer) > 20:
        score -= 0.15

    # 3. Detect formatting issues or artifacts
    if re.search(r’\brn\b|\.{3,}|\.s\.|\s{3,}’, answer):
        score -= 0.1

    # 4. Check for "I don’t have enough information" pattern
    if answer.startswith("I don’t have enough information"):
        # If it’s just that phrase or very little after it
        if len(answer) < 70:
            score -= 0.2
        else:
            # If it has substantial content after the disclaimer
            score -= 0.1

    # 5. Check for question type and expected answer elements
    question_lower = question.lower()

    # "What" questions typically define or explain something
    if question_lower.startswith("what"):
        if re.search(r’is|are|was|were’, question_lower[:15]):
            # Definition question - answer should define the topic
            if re.search(r’(is|are|refers to|defined as|means)’, answer.lower()[:50]):
                score += 0.2

    # "How" questions explain a process or method
    elif question_lower.startswith("how"):
        # Process question - answer should include steps or a method
        if re.search(r’(first|second|then|next|finally|by|through)’, answer.lower()):
            score += 0.2

    # "Where" questions should mention a location
    elif question_lower.startswith("where"):
        # Location question - answer should mention a place
        if re.search(r’(located|at|in|on|near|building|room|floor|campus)’, answer.lower()):
            score += 0.2

    # "Who" questions should mention a person or organization
    elif question_lower.startswith("who"):
        # Person/org question - answer should mention a name or title
        if re.search(r’(staff|faculty|office|center|department|director|coordinator)’, answer.lower()):
            score += 0.2

    # Ensure score is in valid range
    return max(0.0, min(1.0, score))

def generate_qa_pairs(self, urls: List[str], max_pairs_per_url: int = 10) -> List[Dict]:
    """Generate high-quality QA pairs for a list of URLs with adaptive strategies for limited content."""
    # Store original filter method for potential restoration
    if not hasattr(self, ’_original_filter_qa_pairs’):
        self._original_filter_qa_pairs = self.filter_qa_pairs

    # 1. Generate questions
    questions = self.generate_questions_from_documents(urls, max_pairs_per_url)

    # Log progress
    logger.info(f"Generated {len(questions)} questions for {len(urls)} URLs")

    # Check if we have very few questions - adapt strategy if needed
    if len(questions) < 5 and len(urls) > 0:
        logger.warning("Very few questions generated, using adaptive strategies")

        # Generate more general questions that require less specific content
        general_questions = self._generate_adaptive_questions(urls)

        # Add to questions list
        questions.extend(general_questions)
        logger.info(f"Added {len(general_questions)} adaptive questions, total: {len(questions)}")

    # 2. Generate answers
    qa_pairs = self.generate_answers(questions)

    # Log progress
    logger.info(f"Generated {len(qa_pairs)} QA pairs")

    # 3. Filter by quality - adaptive threshold based on content quantity
    if len(qa_pairs) < 5:
        # Set lower quality threshold for limited content
        logger.warning("Few QA pairs generated, lowering quality threshold")
        filtered_pairs = self.filter_qa_pairs(qa_pairs, min_score=0.4)
    else:
        # Use standard threshold
        filtered_pairs = self.filter_qa_pairs(qa_pairs)

    # If we still don't have enough pairs, take the best regardless of threshold
    if len(filtered_pairs) < 3 and len(qa_pairs) > 0:
        logger.warning("Very few QA pairs after filtering, taking best available")
        qa_pairs.sort(key=lambda x: x.get("scores", {}).get("overall", 0), reverse=True)
        filtered_pairs = qa_pairs[:min(5, len(qa_pairs))]

    # Log results
    logger.info(f"Final QA pairs after filtering: {len(filtered_pairs)}")

    return filtered_pairs

def filter_qa_pairs(self, qa_pairs: List[Dict], min_score: float = 0.5) -> List[Dict]:
    """Filter QA pairs by quality scores and remove duplicates, with adaptive threshold for limited content."""
    if not qa_pairs:
        return []

    # Determine if we have limited content
    limited_content = len(qa_pairs) < 5

    # Adjust minimum score based on content availability
    if limited_content:
        adjusted_min_score = min_score * 0.8  # 20% lower threshold for limited content
        logger.info(f"Limited content detected, adjusting quality threshold to {adjusted_min_score:.2f}")
    else:
        adjusted_min_score = min_score

    # Filter by minimum quality score
    quality_pairs = [pair for pair in qa_pairs
                    if pair.get("scores", {}).get("overall", 0) >= adjusted_min_score]

    # If we have very few pairs after filtering, accept lower quality ones
    if len(quality_pairs) < 3 and len(qa_pairs) > 3:
        # Sort by quality and take top 3 regardless of threshold
        sorted_pairs = sorted(qa_pairs, key=lambda x: x.get("scores", {}).get("overall", 0), reverse=True)
        quality_pairs = sorted_pairs[:3]
        logger.info(f"Few high-quality pairs, accepting top {len(quality_pairs)} pairs regardless of threshold")

    # Group by URL
    url_to_pairs = {}
    for pair in quality_pairs:
        url = pair.get("source_url", "")
        if url not in url_to_pairs:
            url_to_pairs[url] = []
        url_to_pairs[url].append(pair)

    # For each URL, deduplicate and select best pairs
    final_pairs = []

    for url, pairs in url_to_pairs.items():
        # Sort by overall score
        pairs.sort(key=lambda x: x.get("scores", {}).get("overall", 0), reverse=True)

        # Select unique pairs (avoid answer duplication)
        unique_pairs = []
        seen_answers = set()

        for pair in pairs:
            answer_key = self._get_answer_signature(pair["answer"])
            if answer_key not in seen_answers:
                unique_pairs.append(pair)
                seen_answers.add(answer_key)

        final_pairs.extend(unique_pairs)

    # Log summary of filtering
    logger.info(f"QA filtering: {len(qa_pairs)} original pairs -> {len(final_pairs)} final pairs")

    return final_pairs

def _get_answer_signature(self, answer: str) -> str:
    """Create a signature for an answer to identify near-duplicates."""
    # Remove stopwords and normalize
    words = [w.lower() for w in answer.split() if w.lower() not in self.resource_manager.stopwords]

    # Sort to make order-independent
    words.sort()

    # Take first 10 words as signature
    signature = " ".join(words[:10])

    return signature

#----------------------------------------------------------------------
# Complete QA Generation System with Checkpointing
#----------------------------------------------------------------------

class QAGenerationSystem:
"""Complete QA generation system with checkpointing and evaluation."""

def __init__(self,
            base_url: str,
            output_dir: str,
            use_gpu: bool = True,
            checkpoint_dir: str = "checkpoints"):
    """Initialize the QA generation system."""
    self.base_url = base_url
    self.output_dir = output_dir
    self.use_gpu = use_gpu
    self.checkpoint_dir = checkpoint_dir

    # Create output and checkpoint directories
    os.makedirs(output_dir, exist_ok=True)
    os.makedirs(checkpoint_dir, exist_ok=True)

    # Create unique ID for this run based on base URL
    self.run_id = hashlib.md5(base_url.encode('utf-8')).hexdigest()

    # Initialize resource manager
    self.resource_manager = ResourceManager()

    # Initialize model manager
    self.model_manager = ModelManager(use_gpu=use_gpu)

    # Initialize knowledge base
    self.knowledge_base = KnowledgeBase(use_gpu=use_gpu)

    # Initialize crawler with longer delay (2.0 seconds instead of 1.0)
    self.crawler = PersistentCrawler(base_url, delay=2.0, checkpoint_dir=checkpoint_dir)

    # Set common paths that should exist on most sites
    self.crawler.priority_paths = [
        "/",
        "/index.html",
        "/about",
        "/contact",
        "/services",
        "/resources"
    ]

    # Initialize document processor
    self.doc_processor = DocumentProcessor(self.resource_manager, self.model_manager, use_gpu=use_gpu)

    # Initialize QA generator
    self.qa_generator = QAGenerator(self.resource_manager, self.model_manager, self.knowledge_base, use_gpu=use_gpu)

    # Track progress
    self.progress = {
        "resources_setup": False,
        "crawling": False,
        "document_processing": False,
        "knowledge_base": False,
        "qa_generation": False,
        "evaluation": False
    }

def _get_progress_path(self) -> str:
    """Get the path for the progress checkpoint file."""
    return os.path.join(self.checkpoint_dir, f"progress_{self.run_id}.json")

def _get_knowledge_base_path(self) -> str:
    """Get the path for the knowledge base checkpoint file."""
    return os.path.join(self.checkpoint_dir, f"kb_{self.run_id}.json")

def _get_qa_pairs_path(self) -> str:
    """Get the path for the QA pairs checkpoint file."""
    return os.path.join(self.checkpoint_dir, f"qa_pairs_{self.run_id}.json")

def save_progress(self) -> None:
    """Save current progress to checkpoint file."""
    try:
        progress_path = self._get_progress_path()

        # Save progress
        with open(progress_path, 'w', encoding='utf-8') as f:
            json.dump({
                "base_url": self.base_url,
                "progress": self.progress,
                "timestamp": datetime.now().isoformat()
            }, f)

        logger.info(f"Progress saved to {progress_path}")
    except Exception as e:
        logger.error(f"Error saving progress: {str(e)}")

def load_progress(self) -> bool:
    """Load progress from checkpoint file."""
    try:
        progress_path = self._get_progress_path()

        if not os.path.exists(progress_path):
            logger.info("No progress checkpoint found")
            return False

        # Load progress
        with open(progress_path, 'r', encoding='utf-8') as f:
            data = json.load(f)

        # Verify base URL
        if data.get("base_url") != self.base_url:
            logger.warning(f"Progress checkpoint is for a different URL: {data.get('base_url')}")
            return False

        # Load progress
        self.progress = data.get("progress", {})

        logger.info(f"Progress loaded from {progress_path}")
        return True

    except Exception as e:
        logger.error(f"Error loading progress: {str(e)}")
        return False

def save_qa_pairs(self, qa_pairs: List[Dict]) -> None:
    """Save QA pairs to checkpoint file."""
    try:
        qa_path = self._get_qa_pairs_path()

        # Save QA pairs
        with open(qa_path, 'w', encoding='utf-8') as f:
            json.dump(qa_pairs, f)

        logger.info(f"QA pairs saved to {qa_path}")

    except Exception as e:
        logger.error(f"Error saving QA pairs: {str(e)}")

def load_qa_pairs(self) -> List[Dict]:
    """Load QA pairs from checkpoint file."""
    try:
        qa_path = self._get_qa_pairs_path()

        if not os.path.exists(qa_path):
            logger.info("No QA pairs checkpoint found")
            return []

        # Load QA pairs
        with open(qa_path, 'r', encoding='utf-8') as f:
            qa_pairs = json.load(f)

        logger.info(f"Loaded {len(qa_pairs)} QA pairs from {qa_path}")
        return qa_pairs

    except Exception as e:
        logger.error(f"Error loading QA pairs: {str(e)}")
        return []

def setup_resources(self) -> bool:
    """Set up all required resources."""
    try:
        # Skip if already done
        if self.progress.get("resources_setup", False):
            logger.info("Resources already set up, skipping")
            return True

        # Set up NLTK resources
        logger.info("Setting up NLTK resources")
        self.resource_manager.setup_nltk()

        # Set up spaCy
        logger.info("Setting up spaCy")
        self.resource_manager.setup_spacy()

        # Set up HuggingFace access
        logger.info("Setting up HuggingFace access")
        self.resource_manager.setup_huggingface_access()

        # Load document processing models
        logger.info("Loading document processing models")
        self.doc_processor.load_models()

        # Load QA generation models
        logger.info("Loading QA generation models")
        self.qa_generator.load_models()

        # Update progress
        self.progress["resources_setup"] = True
        self.save_progress()

        return True

    except Exception as e:
        logger.error(f"Error setting up resources: {str(e)}")
        return False

def crawl_website(self, max_pages: int = 30) -> bool:
    """Crawl website and extract content."""
    try:
        # Skip if already done
        if self.progress.get("crawling", False):
            logger.info("Crawling already done, skipping")
            return True

        # Try to load checkpoint first
        checkpoint_loaded = self.crawler.load_checkpoint()

        if checkpoint_loaded:
            # Validate checkpoint
            self.crawler.validate_checkpoint()

            # If we have enough pages, skip crawling
            if len(self.crawler.content_cache) >= max_pages:
                logger.info(f"Checkpoint loaded with {len(self.crawler.content_cache)} pages, skipping crawl")

                # Update progress
                self.progress["crawling"] = True
                self.save_progress()

                return True

            logger.info(f"Checkpoint loaded with {len(self.crawler.content_cache)} pages, continuing crawl")

        # Crawl website
        logger.info(f"Crawling website: {self.base_url}")
        page_contents = self.crawler.crawl(max_pages=max_pages)

        # Check if we have enough pages
        if not page_contents:
            logger.error("No pages crawled")
            return False

        # If we have very few pages, try to crawl again with different settings
        if len(page_contents) < 3:
```

```python
                logger.warning(f"Only {len(page_contents)} pages crawled, trying again with different settings")

                # Try a different base URL (www. version or non-www version)
                parsed_url = urlparse(self.base_url)
                if parsed_url.netloc.startswith('www.'):
                    new_base = parsed_url.netloc[4:]
                else:
                    new_base = 'www.' + parsed_url.netloc

                new_url = f"{parsed_url.scheme}://{new_base}{parsed_url.path}"

                # Create a new crawler with the alternative URL
                alt_crawler = PersistentCrawler(new_url, delay=3.0, checkpoint_dir=self.checkpoint_dir)
                alt_crawler.priority_paths = self.crawler.priority_paths

                # Try to crawl with the alternative URL
                logger.info(f"Trying alternative URL: {new_url}")
                alt_contents = alt_crawler.crawl(max_pages=max_pages)

                # Merge results if we found more pages
                if len(alt_contents) > len(page_contents):
                    logger.info(f"Alternative URL yielded more pages: {len(alt_contents)}")
                    page_contents = alt_contents
                    self.crawler.content_cache.update(alt_contents)

            # Log results
            logger.info(f"Crawled {len(page_contents)} pages")

            # Save checkpoint
            self.crawler.save_checkpoint()

            # Update progress
            self.progress["crawling"] = True
            self.save_progress()

            return True

        except Exception as e:
            logger.error(f"Error crawling website: {str(e)}")
            return False

    def process_documents(self) -> bool:
        """Process crawled documents into a knowledge base."""
        try:
            # Skip if already done
            if self.progress.get("document_processing", False) and self.progress.get("knowledge_base", False):
                logger.info("Document processing already done, skipping")

                # Try to load knowledge base
                kb_loaded = self.knowledge_base.load(self._get_knowledge_base_path())

                if kb_loaded:
                    logger.info(f"Knowledge base loaded with {len(self.knowledge_base.chunks)} chunks")
                    return True
                else:
                    logger.warning("Failed to load knowledge base, reprocessing documents")

            # Check if we have crawled documents
            if not self.progress.get("crawling", False):
                logger.warning("No crawled documents, run crawl_website first")
                return False

            # Get crawled content
            page_contents = self.crawler.content_cache

            if not page_contents:
                logger.error("No crawled content found")
                return False

            # Process each document
            logger.info(f"Processing {len(page_contents)} documents")

            for url, content in tqdm(page_contents.items(), desc="Processing documents"):
                try:
                    # Extract title from content
                    title_match = re.search(r'TITLE: (.*?)(\n|$)', content)
                    title = title_match.group(1) if title_match else ""

                    # Process document
                    doc = self.doc_processor.process_document(content, url, title)

                    # Add to knowledge base
                    if doc and doc.chunks:
                        self.knowledge_base.add_document(doc)

                except Exception as e:
                    logger.error(f"Error processing document {url}: {str(e)}")

            # Log results
            kb_stats = self.knowledge_base.get_stats()
            logger.info(f"Processed {kb_stats['documents']} documents into {kb_stats['chunks']} chunks")

            # Save knowledge base
            self.knowledge_base.save(self._get_knowledge_base_path())

            # Update progress
            self.progress["document_processing"] = True
            self.progress["knowledge_base"] = True
            self.save_progress()

            return True

        except Exception as e:
            logger.error(f"Error processing documents: {str(e)}")
            return False


    def evaluate_qa_pairs(self, qa_pairs: List[Dict]) -> Dict:
        """Evaluate QA pairs quality metrics."""
        try:
            # Skip if already done
            if self.progress.get("evaluation", False):
                logger.info("Evaluation already done, skipping")
                return {}

            if not qa_pairs:
                logger.warning("No QA pairs to evaluate")
                return {}

            # Calculate various metrics
            metrics = {
                "total_pairs": len(qa_pairs),
                "avg_scores": {},
                "distribution": {},
                "topic_coverage": {},
                "url_coverage": {}
            }

            # Average scores
            score_keys = list(qa_pairs[0].get("scores", {}).keys())
            for key in score_keys:
                avg_score = sum(pair["scores"].get(key, 0) for pair in qa_pairs) / max(1, len(qa_pairs))
                metrics["avg_scores"][key] = round(avg_score, 3)

            # Score distribution
            metrics["distribution"] = {
                "excellent": len([p for p in qa_pairs if p["scores"].get("overall", 0) >= 0.8]),
                "good": len([p for p in qa_pairs if 0.7 <= p["scores"].get("overall", 0) < 0.8]),
                "average": len([p for p in qa_pairs if 0.6 <= p["scores"].get("overall", 0) < 0.7]),
                "below_avg": len([p for p in qa_pairs if p["scores"].get("overall", 0) < 0.6])
            }

            # Topic coverage
            topics = {}
            for pair in qa_pairs:
                topic = pair.get("topic", "").lower()
                if topic and topic != "general":
                    topics[topic] = topics.get(topic, 0) + 1

            metrics["topic_coverage"] = dict(sorted(topics.items(), key=lambda x: x[1], reverse=True)[:10])

            # URL coverage
            urls = {}
            for pair in qa_pairs:
                url = pair.get("source_url", "")
                if url:
                    urls[url] = urls.get(url, 0) + 1

            metrics["url_coverage"] = dict(sorted(urls.items(), key=lambda x: x[1], reverse=True))

            # Save evaluation results
            eval_path = os.path.join(self.output_dir, "evaluation_metrics.json")
            with open(eval_path, ’w’, encoding=’utf-8’) as f:
                json.dump(metrics, f, indent=2)

            logger.info(f"Evaluation metrics saved to {eval_path}")

            # Update progress
            self.progress["evaluation"] = True
            self.save_progress()

            return metrics

        except Exception as e:
            logger.error(f"Error evaluating QA pairs: {str(e)}")
            return {}

    def generate_qa_pairs(self, max_pairs_per_url: int = 10) -> List[Dict]:
        """Generate QA pairs from processed documents."""
        try:
            # Check if we can load from checkpoint
            if self.progress.get("qa_generation", False):
                logger.info("QA generation already done, loading from checkpoint")
                qa_pairs = self.load_qa_pairs()

                if qa_pairs:
                    logger.info(f"Loaded {len(qa_pairs)} QA pairs from checkpoint")
                    return qa_pairs
                else:
                    logger.warning("Failed to load QA pairs, regenerating")

            # Check if we have processed documents
            if not self.progress.get("document_processing", False) or not self.progress.get("knowledge_base", False):
                logger.warning("No processed documents, run process_documents first")
                return []

            # Check knowledge base
            kb_stats = self.knowledge_base.get_stats()
            if kb_stats["chunks"] == 0:
                logger.error("Knowledge base is empty")
                return []

```

```python
            # Get all URLs
            urls = list(set(chunk.doc_url for chunk in self.knowledge_base.chunks))

            if not urls:
                logger.error("No URLs found in knowledge base")
                return []

            # Store original filter function for potential reset
            if not hasattr(self.qa_generator, '_original_filter_qa_pairs'):
                self.qa_generator._original_filter_qa_pairs = self.qa_generator.filter_qa_pairs

            # Generate QA pairs
            logger.info(f"Generating QA pairs for {len(urls)} URLs")
            qa_pairs = self.qa_generator.generate_qa_pairs(urls, max_pairs_per_url)

            # Check if we have enough QA pairs
            if len(qa_pairs) < 5:
                logger.warning(f"Only {len(qa_pairs)} QA pairs generated, trying with lower quality threshold")

                # Lower quality threshold for limited content
                original_filter = self.qa_generator.filter_qa_pairs

                # Override with more lenient filter
                def lenient_filter(pairs, min_score=0.6):
                    return original_filter(pairs, min_score=0.4)

                # Apply the lenient filter
                self.qa_generator.filter_qa_pairs = lenient_filter

                # Try again
                qa_pairs = self.qa_generator.generate_qa_pairs(urls, max_pairs_per_url)
                logger.info(f"After adjustment: {len(qa_pairs)} QA pairs")

                # Restore original filter
                self.qa_generator.filter_qa_pairs = original_filter

            # Save QA pairs
            self.save_qa_pairs(qa_pairs)

            # Update progress
            self.progress["qa_generation"] = True
            self.save_progress()

            # Log results
            logger.info(f"Generated {len(qa_pairs)} QA pairs")

            return qa_pairs

        except Exception as e:
            logger.error(f"Error generating QA pairs: {str(e)}")
            return []

    def save_output(self, qa_pairs: List[Dict]) -> None:
        """Save final output in multiple formats."""
        try:
            if not qa_pairs:
                logger.warning("No QA pairs to save")
                return

            # 1. Save as JSON
            json_path = os.path.join(self.output_dir, "qa_pairs_final.json")
            with open(json_path, 'w', encoding='utf-8') as f:
                json.dump({"qa_pairs": qa_pairs}, f, indent=2)

            # 2. Save as CSV for easy viewing
            csv_path = os.path.join(self.output_dir, "qa_pairs_final.csv")
            with open(csv_path, 'w', encoding='utf-8', newline='') as f:
                # Write header
                f.write("Question,Answer,Source URL,Topic,Topic Type,Overall Score\n")

                # Write data
                for pair in qa_pairs:
                    question = pair["question"].replace(’"’, ’""’)
                    answer = pair["answer"].replace(’"’, ’""’)
                    url = pair.get("source_url", "").replace(’"’, ’""’)
                    topic = pair.get("topic", "").replace(’"’, ’""’)
                    topic_type = pair.get("topic_type", "").replace(’"’, ’""’)
                    score = str(pair.get("scores", {}).get("overall", 0))

                    f.write(f’"{question}","{answer}","{url}","{topic}","{topic_type}",{score}\n’)

            # 3. Save as Markdown for human reading
            md_path = os.path.join(self.output_dir, "qa_pairs_final.md")
            with open(md_path, ’w’, encoding=’utf-8’) as f:
                f.write(f"# QA Pairs for {self.base_url}\n\n")

                # Group by URL
                url_to_pairs = {}
                for pair in qa_pairs:
                    url = pair.get("source_url", "Unknown")
                    if url not in url_to_pairs:
                        url_to_pairs[url] = []
                    url_to_pairs[url].append(pair)

                # Write each URL’s pairs
                for url, pairs in url_to_pairs.items():
                    f.write(f"## {url}\n\n")

                    for i, pair in enumerate(pairs, 1):
                        f.write(f"### Q{i}: {pair[’question’]}\n\n")
                        f.write(f"{pair[’answer’]}\n\n")
                        f.write(f"*Topic: {pair.get(’topic’, ’N/A’)} | "
                              f"Type: {pair.get(’topic_type’, ’N/A’)} | "
                              f"Score: {pair.get(’scores’, {}).get(’overall’, 0):.2f}*\n\n")
                        f.write("---\n\n")

            logger.info(f"Output saved to {self.output_dir} in JSON, CSV, and Markdown formats")

        except Exception as e:
            logger.error(f"Error saving output: {str(e)}")

    def run(self, max_pages: int = 30, max_pairs_per_url: int = 10) -> List[Dict]:
        """Run the complete QA generation pipeline."""
        start_time = datetime.now()

        # 1. Load progress if available
        self.load_progress()

        try:
            # 2. Setup resources
            success = self.setup_resources()
            if not success:
                logger.error("Failed to set up resources")
                return []

            # 3. Crawl website
            success = self.crawl_website(max_pages)
            if not success:
                logger.error("Failed to crawl website")
                return []

            # 4. Process documents
            success = self.process_documents()
            if not success:
                logger.error("Failed to process documents")
                return []

            # 5. Generate QA pairs
            qa_pairs = self.generate_qa_pairs(max_pairs_per_url)
            if not qa_pairs:
                logger.error("Failed to generate QA pairs")
```

```python
                return []

            # 6. Evaluate QA pairs
            self.evaluate_qa_pairs(qa_pairs)

            # 7. Save output
            self.save_output(qa_pairs)

            # 8. Log completion
            end_time = datetime.now()
            duration = end_time - start_time

            logger.info(f"QA generation complete in {duration}")
            logger.info(f"Generated {len(qa_pairs)} QA pairs")
            logger.info(f"Output saved to {self.output_dir}")

            return qa_pairs

        except Exception as e:
            logger.error(f"Error running QA generation: {str(e)}")
            return []
        finally:
            # Clean up resources
            try:
                if hasattr(self, 'model_manager'):
                    self.model_manager.cleanup()
            except Exception as e:
                logger.error(f"Error cleaning up: {str(e)}")

#----------------------------------------------------------------------
# Main Application Entry Point
#----------------------------------------------------------------------

def parse_arguments():
    """Parse command line arguments."""
    parser = argparse.ArgumentParser(description="Generate QA pairs from a website")

    parser.add_argument("--url", type=str, default="https://www.thrive.pitt.edu",
                       help="Base URL to crawl")
    parser.add_argument("--output", type=str, default="qa_output",
                       help="Output directory")
    parser.add_argument("--max-pages", type=int, default=30,
                       help="Maximum number of pages to crawl")
    parser.add_argument("--max-pairs", type=int, default=10,
                       help="Maximum number of QA pairs per page")
    parser.add_argument("--no-gpu", action="store_true",
                       help="Disable GPU usage")
    parser.add_argument("--checkpoint", type=str, default="checkpoints",
                       help="Checkpoint directory")
    parser.add_argument("--force", action="store_true",
                       help="Force regeneration even if checkpoints exist")

    return parser.parse_args()

def main():
    args = parse_arguments()

    print("=" * 80)
    print("Advanced QA Generation System with RAG, Neural Models and Semantic Processing")
    print("=" * 80)
    print(f"URL: {args.url}")
    print(f"Output directory: {args.output}")
    print(f"Max pages: {args.max_pages}")
    print(f"Max QA pairs per page: {args.max_pairs}")
    print(f"GPU enabled: {not args.no_gpu}")
    print("=" * 80)

    try:
        # Create system
        system = QAGenerationSystem(
            base_url=args.url,
            output_dir=args.output,
            use_gpu=not args.no_gpu,
            checkpoint_dir=args.checkpoint
        )

        # If force option is set, clear progress
        if args.force:
            system.progress = {key: False for key in system.progress}
            system.save_progress()
            logger.info("Forced regeneration, cleared progress")


        qa_pairs = system.run(
            max_pages=args.max_pages,
            max_pairs_per_url=args.max_pairs
        )

        print("\nGeneration Summary:")
        print(f"Generated {len(qa_pairs)} QA pairs")
        print(f"Output saved to {args.output}")
        print("=" * 80)

        return 0

    except KeyboardInterrupt:
        print("\nOperation cancelled by user")
        return 1
    except Exception as e:
        print(f"\nError: {str(e)}")
        logger.error(f"Fatal error: {str(e)}")
        traceback.print_exc()
        return 1

if __name__ == "__main__":
    sys.exit(main())
```

10 Code Usage Instructions

10.1 Prerequisites

Before running the QA generation system, ensure you have the following prerequisites:

- Python 3.8 or higher
- CUDA-compatible GPU (recommended) with appropriate drivers
- At least 8GB of RAM (16GB recommended)
- At least 5GB of free disk space
- Internet connection for downloading models and crawling websites

10.2 Installation

Install the required packages using pip:

```
pip install torch transformers beautifulsoup4 nltk spacy tqdm backoff
    sentence-transformers requests numpy sklearn
python -m spacy download en_core_web_sm
```

10.3 Output Directory Structure

```
│ thrive_qa_generator.py ........ Main QA generation script
└─ qa_output/ ........ Created automatically
   ├─ qa_pairs_final.json ........ QA pairs in JSON format
   ├─ qa_pairs_final.csv ........ QA pairs in CSV format
   ├─ qa_pairs_final.md ........ QA pairs in Markdown format
   └─ evaluation_metrics.json ........ Quality metrics for generated QA pairs
└─ checkpoints/ ........ For storing crawl and model state
   ├─ content_*.json ........ Cached webpage content
   ├─ visited_*.json ........ Record of visited URLs
   ├─ failed_*.json ........ Failed URL attempts
   ├─ scores_*.json ........ Page quality scores
   ├─ kb_*.json ........ Knowledge base data
   ├─ qa_pairs_*.json ........ Generated QA pair checkpoints
   └─ progress_*.json ........ Pipeline progress tracking
└─ debug_html/ ........ Optional: Created for HTML debugging
   ├─ page_0.html ........ Sample of downloaded pages
   └─ page_1.html
```

Note: The asterisk (*) in filenames (e.g., `content_*.json`) represents a unique identifier generated for each crawling session, based on an MD5 hash of the website URL. This allows the system to maintain separate checkpoint files for different websites.

For example, when crawling `https://www.thrive.pitt.edu`, the system might create files like:

```
content_a7f92e3b4c.json
visited_a7f92e3b4c.json
kb_a7f92e3b4c.json
```

10.4 Command Structure

```
python thrive_qa_generator.py \
    --url https://www.thrive.pitt.edu \
    --output qa_output \
    --max-pages 30 \
    --max-pairs 10 \
    --checkpoint checkpoints \
    --no-gpu \
    --force
```

10.5 Command-Line Arguments

Argument	Default	Description
`-url`	https://www.thrive.pitt.edu	Base URL to crawl for generating QA pairs.
`-output`	qa_output	Directory to save the generated QA pairs and evaluation metrics.
`-max-pages`	30	Maximum number of pages to crawl from the website.
`-max-pairs`	10	Maximum number of QA pairs to generate per page.
`-no-gpu`	False	Flag to disable GPU usage and run only on CPU.
`-checkpoint`	checkpoints	Directory for saving and loading checkpoint data (crawled content, model states, etc.).
`-force`	False	Force regeneration of QA pairs even if checkpoints exist from previous runs.

Table 5: Command-line arguments for the QA generation script.

References

Ji, Z., Vu, M., Wang, X., and Neubig, G. (2023). A survey of hallucination in large language models. *arXiv preprint arXiv:2303.17085*.

OpenAI (2023). Gpt-4 system card.