Building the Metadata Module for the Content Creation Pipeline

A project report by Group 19

Sponsor: NETC Team

Randy Denney, Franklin Ludgood, Heather Cushnie-Wescott, James Yanello

Name	Email
Mohammad Chowdhury	mchowdhury2020@fau.edu
Isaam Siddique	isiddique2020@fau.edu
Sekai Wynn	swynn2022@fau.edu
Akeno Williams	akenowilliam2016@fau.edu
Dylan Hawryluk	dhawryluk2020@fau.edu

College of Engineering and Computer Science

EGN 4952C - Fall 2024

December 6, 2024

Summary:

The team is developing a system to automate content tagging, build a knowledge graph, and improve metadata organization for training materials.

I. Introduction (Mohammad)

This proposal responds to an RFP from NETC, which seeks to streamline the content tagging process using metadata. NETC is looking for a solution to address the inefficiencies and manual effort in tagging training materials, such as videos, PDFs, Word documents, and audio files. They expect a proof of concept for a system that can automate metadata extraction, enabling efficient and scalable content cataloging for improved organization and retrieval.

A. Background

Training materials such as videos, PDFs, Word documents, and audio files require manual metadata tagging for efficient cataloging and retrieval in the current content creation pipeline. This manual process is time-consuming, prone to errors, and not scalable.

B. Statement of the Problem

The inefficiencies and delays caused by manual metadata tagging hinder the effective distribution of training materials. An automated system that can accurately process and store metadata extracted from various training materials is needed to address these challenges, ensuring timely and organized access to content.

II. Scope of Work

A. Overview (Mohammad)

The team proposes developing a module that leverages SpaCy, an NLP tool for entity extraction, and Meta's LLaMA 2 7B model for generating insights. This module will analyze extracted summary text to identify and extract relationships, which will be stored in a Neo4j graph database. The objective is to construct a comprehensive knowledge graph that captures and visualizes connections between training materials (learner objects) and various physical components. This approach will streamline the cataloging, retrieval, and understanding of content and its relationships, greatly enhancing the overall management of training resources.

To do this, the team must establish communication between the various components involved in the module's operation. The first step is to create a status feed to build and publish messages to the dashboard queue, providing live status updates from all components. Key components include the parser module, which will check the queue for messages and store files in a designated folder or bucket based on a unique Content ID, keeping all document elements organized.

Once the files are stored, the document module will send the summary text as an array of sentences to the analyzer module. It will then notify the status feed that the files have been saved and are being sent to the analyzer for relationship extraction or send the files to the data error component if processing is impossible. The analyzer will process the text, extract relationships, and send them to the graph database function. A message will then be sent to the status feed, confirming that the analysis is complete and the data is being sent to build the knowledge graph.

The database operation will take inputs in the form of string arrays (e.g., ["node1, relation, node 2"]). It will extract nodes and relationships, perform operations such as checking for existing nodes and relationships, updating or creating them, and then store them in the graph database. Once the knowledge graph is built, a final message will be sent to the status feed, providing more details about the completed knowledge graph.

B. Literature Review (Dylan & Mohammad)

In the field of metadata management, traditional methods often rely on predefined standards and schemas, which may not adequately handle unstructured data from diverse sources such as images and documents. As a result, there has been a shift towards more dynamic and context-sensitive techniques, particularly leveraging advancements in machine learning and database management technologies.

Overview of Machine Learning Techniques

Machine learning (ML), including natural language processing (NLP) and computer vision, plays a pivotal role in modern metadata management. For this project, we chose SpaCy, a powerful tool for NLP tasks due to its robust ability to analyze language patterns and structures.

SpaCy will be used to extract specific terms and key phrases from text, such as “F16,” “Wings,” and “Engine.” These extracted terms will serve as nodes and edges, representing relationships between entities within a graph database. This graph structure will enhance metadata accessibility, allowing users to retrieve relevant information efficiently. Computer vision techniques can also be applied to analyze images, identifying objects and features that further enrich the metadata. This knowledge graph can then be leveraged to build proprietary MLMs.

Algorithm Development and Database Management

Developing algorithms to process and retain relevant data involves a combination of supervised and unsupervised learning methods. These algorithms can filter out irrelevant information, focusing on metadata that accurately reflects the content context. For storing this metadata, graph databases like Neo4j are often used. Unlike traditional relational databases, graph databases are optimized for handling complex relationships between data points, making them ideal for scenarios where the connections between different data elements are as important as the data itself.[1]

Comparative Analysis

Compared to traditional methods that require manual data categorization and rely on fixed schemas, modern approaches using ML and graph databases offer enhanced accuracy and flexibility. They can automatically adapt to new data and contexts, thus reducing the workload and potential for human error. However, these approaches also present challenges, such as ensuring the quality and representativeness of training data and managing the computational demands of machine learning models.

Applications and Implications

Integrating machine learning and advanced database management systems is applicable across various domains, including content management, technical documentation, and more. These technologies improve the efficiency of search functions and enhance the user experience by providing more precise and contextually relevant information. For example, in industries like aerospace, this system could streamline the management of technical manuals and documentation by dynamically organizing data based on its content.[1][2]

Future Directions

Future advancements may include more sophisticated deep learning models, providing even more precise metadata extraction and categorization. Additionally, integrating these systems with technologies like cloud computing and big data analytics could offer scalability and support real-time data processing, further enhancing their utility in large-scale applications.[2]

C. Alternative Solutions (Isaam)

In developing our metadata module, we initially considered a cloud-based machine learning approach using services like Amazon SageMaker or Google Cloud AI Platform. This option offered scalability, pre-trained models, and managed infrastructure, allowing for rapid prototyping and deployment. However, concerns about latency, data privacy, long-term costs, and customization limitations led us to explore alternatives.

Ultimately, we chose a locally implemented solution using the Spacy and Meta llama model and Neo4j database. This approach provides greater control over model fine-tuning, better integration with our existing systems, and full data control. While it requires more initial setup and computational resources, it offers superior customization for our specific metadata extraction and management needs.

D. Evaluation (Akeno)

The evaluation of the metadata module will primarily focus on assessing the accuracy and efficiency of relationship extraction using SpaCy, which has been fine-tuned on a small, custom-built dataset for our project's specific requirements. The main evaluation criteria are technical performance (45%), cost (20%), ease of implementation (20%), and maintainability (15%). Technical performance is prioritized to ensure the system meets sponsor requirements with high accuracy in identifying edges and nodes for constructing a graph database.

Cost is the second most important factor, given our financial constraints. We aimed to deliver a more cost-effective product, as API-based solutions like ChatGPT would have incurred significant costs with increased document processing. SpaCy also fits well within our hardware

limitations, allowing us to host and run the model locally. This ensures seamless integration with existing infrastructure and minimizes latency without requiring external cloud services.

In the coming weeks, we plan to test the fine-tuned SpaCy model on various text documents to measure its effectiveness in identifying entities and relationships. These tests will provide insights into the model's performance and guide further fine-tuning if necessary. Additionally, we will evaluate the integration of the extracted nodes and edges into the graph database to ensure smooth incorporation into our system. This step will also provide valuable insights into response times and resource usage, helping us project future costs and assess maintainability.

Utilizing a local machine learning model with SpaCy allows us to scale flexibly by upgrading hardware as needed, avoiding potential constraints or variable fees associated with cloud-based infrastructure. Future evaluations will measure the system's performance with larger data sets, including complex content types such as multiple images, audio, and text files. This thorough evaluation ensures users that our solution is reliable and efficient in meeting their data processing and storage needs.

E. Decision (Dylan)

Our team's decision to implement a local setup using SpaCy alongside a Neo4j database was made carefully and thoughtfully. The primary factor driving this decision was performance. A cloud-based solution would consume extensive resources and incur high long-term costs, while local hosting ensures greater speed and responsiveness, as it eliminates throttling caused by network latency. Maintaining a local system also keeps costs lower over time, as an established setup requires less ongoing financial investment. Although the initial setup might involve higher upfront costs, these expenses are expected to deliver long-term value.

Security was another significant factor influencing our choice of a local implementation for SpaCy and Neo4j. Since our system will manage confidential information entrusted by our sponsors, minimizing the transfer of sensitive data over the Internet enhances data security. Using a local model provides customization, more direct management, and effective integration,

enabling SpaCy to deliver fast and precise metadata analysis, while Neo4j efficiently organizes data through relationships and nodes.

Despite potentially higher initial costs, the local solution offers superior performance, flexibility, and security, making it the optimal choice to meet our sponsors' objectives and safeguard their data effectively.

III. Implementation Details

A. System Specifications and Functionalities (All Members)

Metadata Ingestion

The system handles metadata and visualizes relationships using a graph database powered by knowledge graphs. The Document Module, which extracts summaries, keywords, and metadata from files, has been enhanced with a new function. This function provides the summary as an array of sentences, enabling more granular processing and analysis within the Metadata Module.

The Analyzer Module processes these sentences to generate a Mission Profile for the main node associated with the content. It builds nodes and relationships based on the extracted information and passes them to the Database Component, which stores them in a Neo4j graph database.

A File Parser Module manages the flow of BSON data and files received from a queue. It organizes files by grouping them under unique Content IDs, ensuring structured local storage. The parser also communicates job progress through a Status Feed, maintaining transparency and enabling real-time updates across the system.

By integrating these components, the system ensures seamless metadata management, efficient relationship visualization, and robust data organization within a graph database.

Metadata Analysis

The data analyzer module performs metadata analysis and serves as a cornerstone of the system, enabling the identification, categorization, and linking of objects to their respective training content. The module transforms unstructured data into a structured knowledge graph by leveraging advanced natural language processing (NLP) techniques and machine learning models. This knowledge graph represents the relationships between key entities, offering a scalable solution for organizing metadata.

Metadata extracted from textual documents, such as maintenance manuals, technical guides, and training content, undergoes processing through distinct functions within the data analyzer module. Each function is optimized for a specific task, such as extracting entities, identifying nodes, and determining relationships based on the identified entities. The outputs—nodes and relationships—are mapped to a Neo4j graph database, where nodes represent objects like engines, aircraft, and training documents. The edges define meaningful relationships such as `has_engine` or `engine_of` This structured approach enables efficient knowledge graph querying, visualization, and expansion.

To extract domain-specific entities precisely, we trained a SpaCy Named Entity Recognition (NER) model from scratch, starting with spacy.blank("en"). This blank pipeline ensured the model had no pre-existing biases, enabling it to focus solely on the entities relevant to our domain. Key entities identified include `digitalTwinAircraft`, `digitalTwinEngine`, `digitalTwinGround`, `digitalTwinMarine`, and `digitalTwinElectricGenerator`.

The training process utilized a dataset of 36 annotated sentences from 3 documents we created containing explicit mentions of entities. Each sentence was annotated to establish clear associations between terms and their corresponding entity labels. For instance:

- In the sentence "The GE414 engine powers the F18 aircraft," the term "GE414" was labeled as `digitalTwinEngine`, while "F18" was labeled as `digitalTwinAircraft`.

This explicit annotation ensured the model learned accurate and domain-relevant mappings between input text and entity labels. The dataset was small but sufficient for rapid testing and validation, forming a strong foundation for subsequent expansions.

Training the model efficiently was critical to ensure timely outputs that could drive downstream processes, including node and relationship storage, message handling, dashboard updates, and file retrieval. The training pipeline incorporated the following components:

- Dynamic Batching: Batch sizes began at 4 and gradually increased to 32 using a growth factor of 1.001. This ensured efficient memory usage and stable training progress.
- Dropout Regularization: A dropout rate of 0.5 was applied during training to mitigate overfitting and improve generalization to unseen data.
- Optimization Algorithm: SpaCy's Adam optimizer was used, initialized through `nlp.begin_training()`. The optimizer employed a learning rate scheduler to dynamically adjust the learning rate, ensuring stable convergence.
- Training Iterations: The model was trained for 100 iterations, allowing sufficient time for the weights to converge and the loss function to stabilize.

Loss values, which began at approximately 319.86 due to the blank start, steadily decreased over the course of training, reaching a final loss of ~3.36. This significant decline reflected the model's ability to effectively learn and adapt to recognizing domain-specific entities. Once trained, the NER model was saved as `custom_ner_modelREL` for reuse. During runtime, the data analyzer module loads this trained model to extract entities consistently and accurately across various documents. These entities form the foundation for constructing nodes and relationships within the graph database.

The system uses a dictionary-based approach to map relationships between identified entities, ensuring consistency and relevance in the knowledge graph. Relationships are determined based on the types of entities and their logical connections, as defined in a relationship map. For example, relationships like (`digitalTwinAircraft` and `digitalTwinEngine`) are mapped to the semantic connection "engine," reflecting their functional association.

During analysis, the system processes sentences using a trained SpaCy NER model, extracting entity labels and checking for predefined relationships between nodes. Each identified relationship is stored as an edge in the Neo4j graph database, with nodes representing the entities and edges defining the type of connection. For instance, if two consecutive entities are `digitalTwinMarine` and `digitalTwinElectricGenerator`, their relationship is defined as "generator," linking the marine entity to its associated electric generator.

This dictionary-based approach allows for efficient parsing and dynamic relationship extraction, ensuring the resulting graph is accurate and contextually meaningful. Future enhancements may expand the relationship map to handle more complex connections and implicit relationships between nodes, improving the system's capacity for capturing intricate metadata structures.

The primary node represents the core subject of the document and serves as the closest node to the learner object, directly linked through a defined relationship. This node is determined based on the frequency of explicit mentions within the text, ensuring that the most relevant and dominant entity is selected. This frequency-based approach provides an efficient and straightforward method for identifying primary nodes, offering a practical interim solution for structured relationship extraction.

For example, in a document discussing the GE414 engine, the primary node would be "GE414," as it is the focus of the content and forms the central link to associated learner objects. While the current implementation relies on explicit mentions, future enhancements can incorporate advanced NLP techniques and larger datasets to handle implicit relationships. This enables the system to identify contextually significant nodes even when explicit mentions are limited.

The system integrates a pre-trained LLaMA2-7B model to complement explicit relationships in the graph and generate mission profiles for the primary node. These profiles are generated by passing extracted summary sentences and a prompt structured as: “What is the {mainNode} and what does it do? Two sentences max.”

For example, if the main node is "GE414," the system produces a concise summary such as:

- "The GE414 is a high-speed computing engine developed by General Electric (GE) for use in various applications, including weather forecasting, scientific simulations, and data analytics. The GE414 engine offers enhanced performance and scalability compared to its predecessor, the GE350…"

This generative approach ensures mission profiles provide contextual insights into the primary node's purpose and functionality. While LLaMA2-7B was initially evaluated for relationship extraction, its generative nature proved more suitable for creating descriptive and informative mission profiles. These summaries enrich the knowledge graph by offering end-users a deeper understanding of each node's significance within the system. Future iterations may expand on this functionality to include dynamic prompts and multi-faceted contextual details.

Integrating the SpaCy NER model, relationship mapping, and mission profile generation creates a cohesive metadata analysis pipeline. This pipeline transforms unstructured metadata into a structured knowledge graph, enabling:

- Accurate extraction of entities and relationships.
- Efficient storage and retrieval of metadata.
- Scalable visualization of interconnected objects.

The system underwent iterative testing using a small dataset, which validated its ability to extract and map relevant entities and relationships accurately. Testing provided insights into potential enhancements and ensured that outputs met operational requirements. For example:

- Refinements in SpaCy fine-tuning improved entity recognition accuracy.
- Adjustments to relationship mapping enhanced edge consistency in the graph database.

The system is designed with scalability in mind. Expanding the training dataset and incorporating more sophisticated NLP techniques will improve the system's ability to handle implicit mentions, complex relationships, and larger document sets. These advancements will ensure the system remains adaptable to evolving use cases and growing data demands.

Data Storage

The Neo4j metadata management system processes and stores relationships between nodes in a structured graph database. The implementation focuses on parsing data, establishing relationships, and providing clear outputs of the stored relationships.

The `addLearnerRelation` function takes two node arrays and a relationship, storing a learner object node and its connection to another node. It uses Neo4j's `MERGE` command to avoid duplicates. For example, it creates a learner object node with properties like name, location, and content ID, while also linking it to a second node with properties such as a mission profile. The function forms the relationships:

- `node1 -[has_relation]-> node2`
- `node2 -[relation_of]-> node1`

The `addDigitalTwinRelation` function works similarly, but focuses on connecting digital twin nodes to other nodes. Both functions ensure the relationships are bidirectional by creating `has_relation` and `relation_of` links.

The `addImageLearner` function creates image-related learner nodes, adding metadata such as content ID, location, and a predicted class for classification purposes. It stores the node in Neo4j and links it to the main content node. A status feed logs the classification and storage operation.

The `packageParser` function is central to processing data packages. It extracts elements such as node arrays and relationships, calling the appropriate add functions to store nodes and relationships in Neo4j. It iteratively processes all relationships in the package and, once completed, calls nodeTraceback to trace and validate relationships for a learner object.

The `nodeTraceback` function traces specific relationships, such as `learnerObject_of`, in one direction. Using a Cypher query, it retrieves the learner object and its connections, formats the relationships, and ensures deduplication before outputting them. For example, if tracing the node "F18GeEngineWithPic.pdf," it outputs the connections:

- `F18GeEngineWithPic.pdf -[learnerObject_of]-> GE414 engine`
- `GE414 engine -[engine_of]-> F18 aircraft`

The `nodeTracebackManual` function provides an interactive method for users to explore nodes by partial names. After selecting a node, it retrieves and displays all relationships connected to that node. The `getAllNodes` function retrieves and prints up to 100 nodes from Neo4j for debugging or verification.

These functions work together to manage and visualize relationships in a Neo4j graph database efficiently. They handle a variety of tasks, from storing metadata to tracing and validating connections, ensuring a clear and organized graph representation of the data.

Files are stored locally, and each entry is tagged with the associated learner object, with the learner object node in the graph database including the local file path for retrieval. The system utilizes RabbitMQ message processing to handle Store, Document, and Image queues, where messages are strictly validated using ContentID matching. When processing messages, the system creates dedicated directories named `store_<contentID>`, where files are organized based on their type. Store messages result in multiple file types including `payload.pdf`, `meta.txt`, `summary.txt`, and `keywords.txt`, while image messages are saved with the format of `image_filename+contentID.png`. The system implements a careful message acknowledgment strategy: successful message processing with matching ContentIDs receives an acknowledgment (`basic_ack`), while messages with non-matching ContentIDs receive no acknowledgment, allowing them to be requeued implicitly. For error scenarios, the system uses `basic_nack` with `requeue=True` for recoverable errors (like processing errors) and `requeue=False` for unrecoverable errors (like malformed BSON messages), ensuring proper message queue management.

In future implementations, this content can be stored in a cloud-based storage bucket, with the learner object node containing the file's bucket location rather than a local path. This approach will enhance scalability and accessibility, supporting efficient categorization and retrieval of related information in a centralized manner. The system implements comprehensive error handling for BSON decoding and file operations, with detailed logging to track processing status and potential issues.

This functionality is integral to building a knowledge graph from various training documents and media, enabling the identification of relationships between training components. By connecting these relationships, the system provides a well-organized and accessible structure for tracking and managing training content, maintaining data organization through ContentID-based structuring while supporting future scalability needs.

Progress Tracking

Progress updates are sent to the dashboard module using RabbitMQ for inter-module communication. The system implements a status feed module that is invoked by other components (data parser, analyzer, database module, and error module) to provide status updates. It uses a `messageBuilder` function that builds BSON messages containing job ID, content ID, status, timestamp, and details. These messages are then sent to RabbitMQ through a publisher function.

Error Handling and Logging

For the error handling and logging portion of the module, we did not have sufficient time or the immediate necessity to prioritize its development. As the project progressed, we identified other critical areas that demanded our attention to ensure the overall functionality and efficiency of the metadata module. These decisions allowed us to focus on delivering a working proof of concept within our constraints.

Had we developed the error handling and logging functionality, it would have been designed to ensure the system's smooth operation and to handle issues in a predictable and manageable manner. This would have started with robust error detection mechanisms that identify potential failure points and capture detailed reasons for those failures. These detected errors would be logged and sent to a status feed for visibility and tracking. Furthermore, unprocessed messages would have been returned to the queue to allow reprocessing, while error-specific messages would also be injected back into the queue to ensure traceability and resolution.

The error handling module would have been closely integrated with the data parser to create a cohesive system. Whenever the parser encountered unexpected data or behavior, it would invoke the error module to analyze the issue, log relevant details, and update the status

feed with comprehensive information about the problem. This approach would not only prevent the parser from silently failing but would also maintain transparency and accountability for any issues encountered. Such a system would ensure no errors went unaddressed, facilitating optimal teamwork between the parser and the error handling module while maintaining system reliability and efficiency.

B. Block Diagram (Sekai & Mohammad)

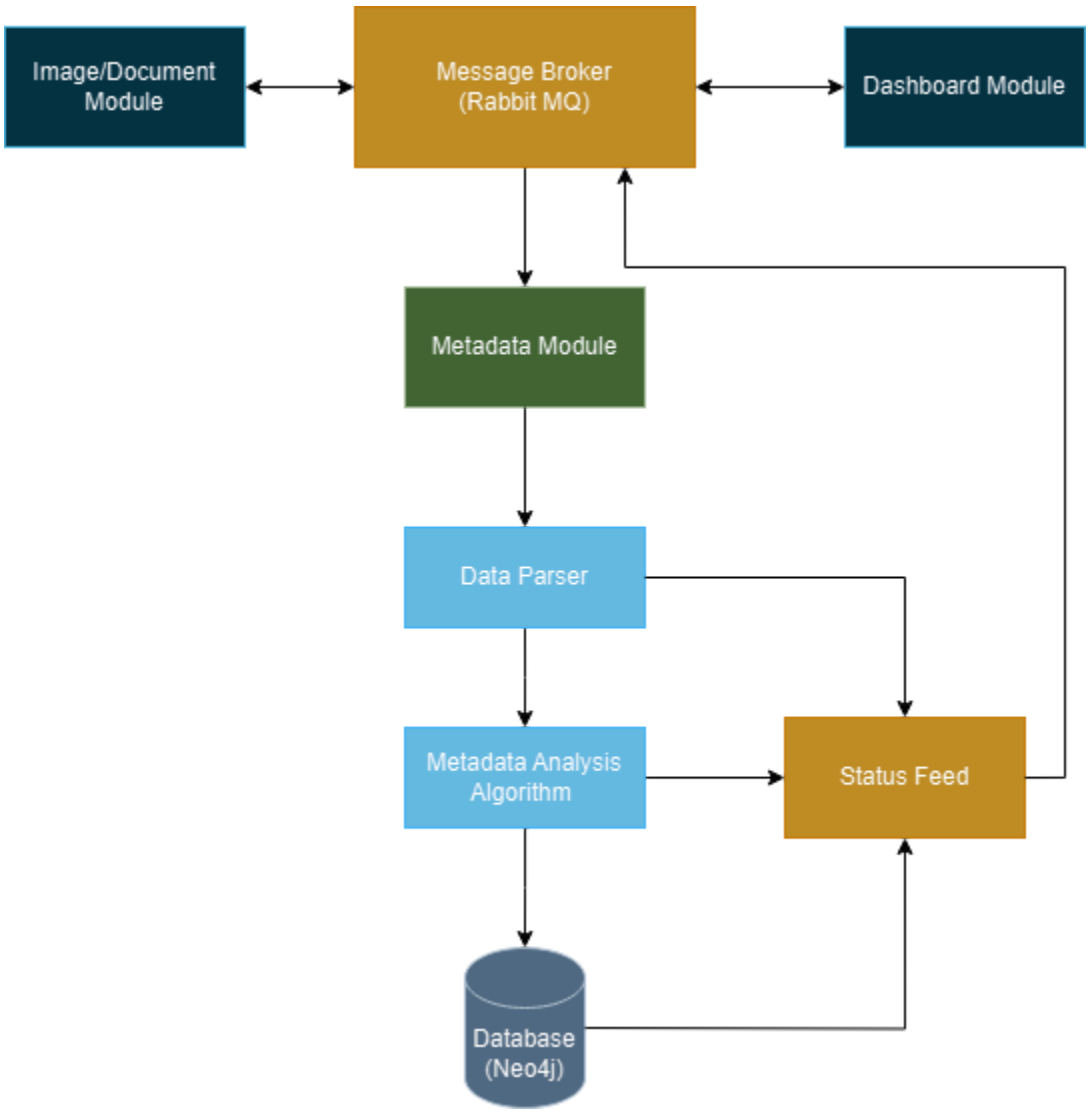

Figure 1 - Block Diagram

C. Flowchart (Sekai & Mohammad)

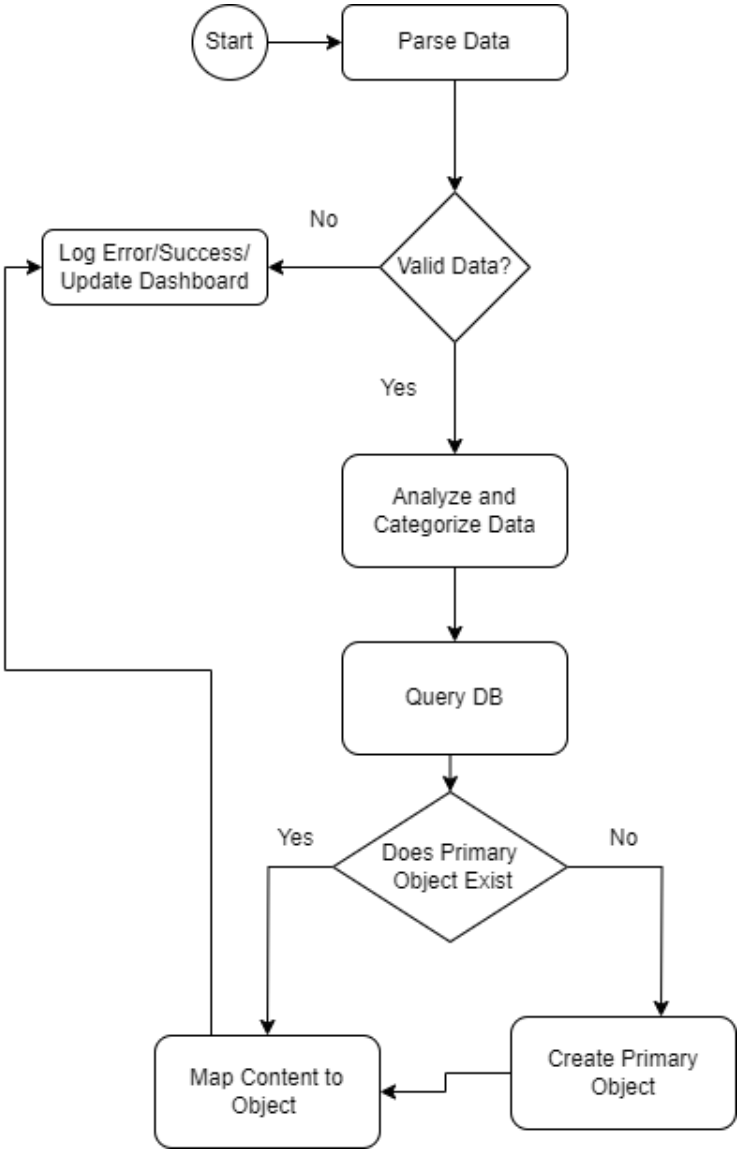

Figure 2 - Flowchart

D. Technology and Standards Used (All Members)

Neo4J (Cypher Query Language)[3]

- Standards Body: OpenCypher Project (Not formally standardized, but based on open source)
- Standard Identification: N/A (Cypher is a de facto standard within the Neo4j ecosystem)
- Standard Name: Cypher Query Language
- Purpose: Neo4j and the Cypher query language are used for graph database management.

RabbitMQ[4]

- Standards Body: OASIS (Organization for the Advancement of Structured Information Standards)
- Standard Identification: AMQP 1.0
- Standard Name: Advanced Message Queuing Protocol (AMQP)
- Purpose: AMQP is a protocol used by RabbitMQ to enable reliable, message-based communication.

WebSockets[5]

- Standards Body: IETF (Internet Engineering Task Force)
- Standard Identification: RFC 6455
- Standard Name: The WebSocket Protocol
- Purpose: WebSockets enable two-way communication between the client and server in real-time applications.

Python[6]

- Standards Body: PSF (Python Software Foundation)
- Standard Identification: PEPs (Python Enhancement Proposals); PEP 8 for style guide
- Standard Name: Python Programming Language
- Purpose: Python is the main programming language used for building the application.

SpaCy (Natural Language Processing Model)[7]

- Standards Body: N/A (Open source software, no formal standards)
- Standard Identification: N/A
- Standard Name: N/A (Model-based usage; follows NLP standards based on the task it's used for like tokenization or Named Entity Recognition)
- Purpose: SpaCy is used for natural language processing to extract relationships from text.

Meta Llama 2 7B (Machine Learning Model)[8]

- Standards Body: N/A (Meta's own proprietary model)
- Standard Identification: N/A
- Standard Name: LLaMA (Large Language Model Meta AI)
- Purpose: LLaMA 2 is used for generating or analyzing text, extracting context-aware relationships.

Flask (Web Framework)[9]

- Standards Body: PSF (Python Software Foundation)
- Standard Identification: N/A (Follows general Python standards like PEP 333 for WSGI)
- Standard Name: N/A (Flask is a framework under Python's umbrella)
- Purpose: Flask is used for building the web server and APIs.

HTTP (for Flask)[10]

- Standards Body: IETF
- Standard Identification: RFC 2616 (for HTTP/1.1)
- Standard Name: Hypertext Transfer Protocol (HTTP)
- Purpose: HTTP is the protocol Flask uses to communicate over the web. Makes API calls.

E. Testing and Performance Evaluation (Mohammad)

Testing included verifying how the system handled duplicate uploads. Our testing began with a small dataset and three PDF documents to validate that the program correctly extracted

the entities we cared about. Specifically, we focused on identifying entities such as engines, aircraft, marine, ground, and generators, collectively labeled as digital twins.

We experimented with three pre-trained BERT models(Bert, Roberta, DistilBert) and a pre-trained SpaCy model. However, these models extracted all named entities indiscriminately, including irrelevant ones, which did not meet our requirements. After three weeks of testing and refinement, we trained a blank SpaCy model from scratch, customizing it to recognize only the entities we were interested in.

We curated a dataset of approximately 40 sentences for training. Each sentence was carefully marked up to highlight the entities we wanted to extract. For example:

- "The GE414 engine is used in the F18 aircraft."
    - Here, "GE414" was labeled as an engine, and "F18" as an aircraft.
- "The M551 tank is equipped with a high-performance generator."
    - In this case, "M551" was labeled as ground, and "generator" as itself.

Training began with a blank SpaCy pipeline, `(spacy.blank("en"))`, to eliminate noise from irrelevant pre-trained entities. Key training parameters included dynamic batching (batch sizes starting at 4 and increasing to 32), dropout regularization (rate: 0.5), and 100 training iterations. The model's loss value decreased from an initial ~319.86 to a final ~3.36, demonstrating effective learning and adaptation. The resulting model was saved as `custom_ner_modelREL` for reuse, enabling consistent and accurate extraction of domain-specific entities during runtime.

Once trained, the NER model was saved as `custom_ner_modelREL` for reuse. During runtime, the data analyzer module loads this trained model to extract entities consistently and accurately across various documents. These entities form the foundation for constructing nodes and relationships within the graph database.

This manual annotation allowed us to precisely define the boundaries and labels for the desired entities, providing the training model with clear and concise examples. With this approach, we ensured the model was tailored to our domain-specific requirements, effectively avoiding noise from irrelevant entities.

In addition to achieving accurate entity extraction, our goal was to train this model quickly to integrate it seamlessly into the rest of the system. The outputs from this model—accurately identified entities—were essential for generating nodes and relationships in our graph database. These nodes and relationships would then be stored, a message would be sent to the dashboard indicating the operation's status, and files would be retrieved for further processing.

When testing how the system handled duplicate uploads, we found that it consistently avoided creating duplicate digital twin nodes in the database. Instead, only learner object nodes were recreated, each with a unique content ID. This was achieved using SHA-256 hashing and timestamps, ensuring that even documents with identical names were uniquely identified and tracked. This approach maintained the integrity of the Neo4j database, as digital twin nodes remained singular while allowing for multiple versions of learner objects to exist.

Additionally, we validated database integrity by running queries to confirm that no redundant nodes or relationships were introduced. This ensured that the digital twin nodes, such as engines or aircraft, were stored uniquely, while learner object nodes, representing uploaded documents, were correctly linked to their respective digital twins. The system traced each learner object to determine the primary digital twin node it was associated with, ensuring accurate and logical node relationships within the Neo4j graph. For example, a learner object like a training PDF was consistently traced back to its related digital twin node, such as "GE414 Engine" or "F18 Aircraft," maintaining the integrity and structure of the graph database.

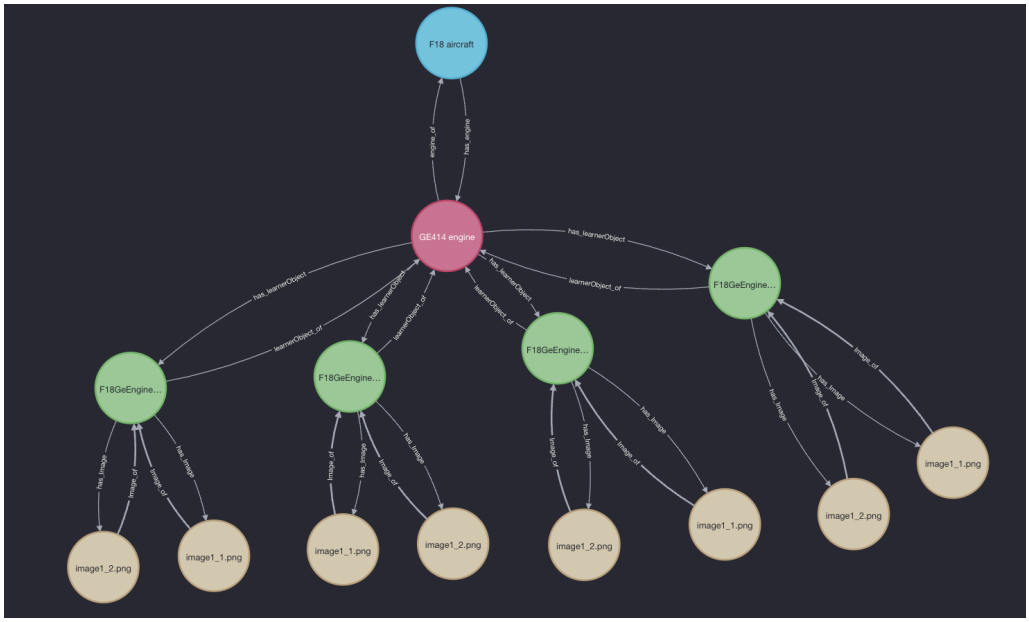

Figure 3 - No Main Node Redundancy

The database testing demonstrated the robustness of the module, handling duplicate and sequential uploads without errors. By ensuring proper tracing of learner objects, accurate node associations, and effective status updates, we established a solid foundation for the metadata module. This approach supports scalability, enabling the system to handle larger datasets and more complex relationships with reliability and efficiency.

As part of the testing process, we verified the status message pipeline to ensure seamless communication between components. Once the learner objects and relationships were successfully stored in the database, a message was sent to the dashboard, confirming the operation's status. This included verifying that the dashboard received all expected messages, such as completion of node creation and file processing. These status updates provided real-time feedback, helping us monitor the system’s performance and ensure every component functioned cohesively.

F. Discussion and Lessons Learned (All Members)

The primary goal of this project was to deliver a proof of concept for the metadata module, which we achieved using a small custom model tailored to meet our specific

requirements. Despite their capabilities, pre-trained models did not align with the task objectives. Developing a working prototype allowed us to identify and document flaws in the system, which will provide valuable insights for future teams working with NETC.

Testing pre-trained BERT models presented significant challenges. Over three weeks, we evaluated three to four models, but training them was beyond our team's expertise and hardware capabilities. Limited CPU processing power and a lack of experience in machine learning hindered our ability to fine-tune these models effectively. Neo4j also posed challenges, as none of the team had prior experience with the platform. This meant that data storage and querying required additional time to learn and implement.

Another hurdle was understanding and adapting the previous team's codebase, which involved deciphering its structure and logic before extending it. Adjustments to the existing codebase were necessary to accommodate new functionality and ensure compatibility with our enhancements. While we succeeded in integrating these features, the process highlighted the importance of clear documentation and modular design for future scalability.

Large-scale testing remains a limitation, as the proof of concept relies on a small, controlled dataset. Evaluating the system's scalability and robustness would require more diverse and extensive data, which was not feasible within the constraints of this project. Additionally, integrating with the dashboard and image teams consumed two weeks of our timeline. This reduced our flexibility, limited time for our own module, and restricted our ability to make dynamic changes closer to project deadlines.

Despite these challenges, the project provided valuable learning opportunities in ML model development, Neo4j integration, and collaborative module design. These experiences, combined with our documented insights, will aid the next group in overcoming similar hurdles and advancing the system further.

IV. Conclusion

The metadata module project successfully delivered a functional proof of concept, demonstrating the feasibility of automating metadata extraction, relationship mapping, and graph database integration for content tagging. The system provides a streamlined approach to managing and organizing training materials by extracting key entities, establishing relationships, and storing them in a graph database for easy retrieval and visualization. The use of a custom model tailored to project requirements allowed the module to effectively meet the specific needs of NETC, despite the limitations of pre-trained models.

This prototype showcases the system's ability to structure and categorize content dynamically, addressing the challenges of data organization and enhancing the accessibility of information. While certain limitations, such as the need for a more diverse dataset and scalability testing, were identified, the current system provides a strong foundation for future enhancements. Detailed documentation of the system’s architecture, workflows, and challenges ensures that future teams can build upon this work to refine and expand its capabilities, moving towards a more robust and scalable solution for content tagging and knowledge graph construction.

V. References

[1] Zhong, H., Yang, D., Shi, S., Wei, L., & Wang, Y. (2024). From data to insights: the application and challenges of knowledge graphs in intelligent audit. Journal of Cloud Computing Advances Systems and Applications, 13(1). https://doi.org/10.1186/s13677-024-00674-0

[2] DeepLearning.AI. (2023, January 11). Natural Language Processing (NLP) [A complete guide]. https://www.deeplearning.ai/resources/natural-language-processing/

[3] Enzo. (2024, November 27). OpenCypher will pave the road to GQL for Cypher implementers. Graph Database & Analytics. https://neo4j.com/blog/opencypher-gql-cypher-implementation/

[4] Rabbitmq. (n.d.). GitHub - rabbitmq/amqp-0.9.1-spec: AMQP 0-9-1 specification-related documents. GitHub. https://github.com/rabbitmq/amqp-0.9.1-spec

[5] Fette, I., & Melnikov, A. (2011). The WebSocket protocol. https://doi.org/10.17487/rfc6455

[6] PEP 8 – Style guide for Python code | peps.python.org. (n.d.). Python Enhancement Proposals (PEPs). https://peps.python.org/pep-0008/

[7] Trained Models & Pipelines · SPACY Models Documentation. (n.d.). Trained Models & Pipelines. https://spacy.io/models

[8] Meta. (2023, February 24). Introducing LLaMA: A foundational, 65-billion-parameter language model. Ai.facebook.com. https://ai.meta.com/blog/large-language-model-llama-meta-ai/

[9] Welcome to Flask — Flask Documentation (3.1.x). (n.d.). https://flask.palletsprojects.com/en/stable/

[10] HTTP - Hypertext Transfer Protocol Overview. (n.d.). https://www.w3.org/Protocols/

VI. Personnel

The entire team consists of computer science majors. The following titles are based on primary contributions:

Team Leader: Mohammad Chowdhury

- System Architect & Integrator

Team Member: Isaam Siddique

- File System & Deployment Specialist

Team Member: Sekai Wynn

- Database Specialist

Team Member: Akeno Williams

- Algorithm Programmer

Team Member: Dylan Hawryluk

- ML Model Data Trainer