Metadata Module Documentation

Overview

This overview will discuss Group 19's Metadata Module, and briefly explain the entire system. This includes the message broker system, what we inherited from the previous team, and our progress. We also aim to highlight some integrations that were completed with the dashboard and image module teams.

First, let's talk about the key files involved in the message broker system. The original files we modified include `parse.py`, `main_server_GUI_socket` (originally called `main_server`), and the `document_module` now called doc_module.py

The message broker system, which uses RabbitMQ, serves as a communication bridge between different software components. If one component needs to send data to another, it does so by transmitting it through WebSockets to a designated file, which then passes it to RabbitMQ. RabbitMQ creates a queue that other software components can access. In essence, the message broker system facilitates communication between software components, though it can also be utilized for other purposes. For our project, the message broker system manages the processing of uploaded documents, audio, images, and videos. Files are uploaded into the queue and remain there until processed.

When we took over the project, the system allowed for uploading five test files—audio, video, PDFs, and similar formats. Upon running the docmodule.py file, the system would process the uploaded files, extract summaries, and retrieve images. The previous team also developed receivers that clear the queue by retrieving all files and their processed information. This is the foundation we started with as we continued development.

Metadata Module

Building upon the foundation provided to us, we were tasked with developing the metadata module. This module is designed to take document summaries, analyze them, identify domain-specific entities, and store those entities in a graph database while mapping their relationships. For example, if a PDF discusses maintenance for the GE414 engine and includes references to the F18 aircraft

and other components, the ideal node relationship would look like this: *F18 has engine GE414,* and *GE414 has learnerObject -> trainingPDF.*

Throughout this project, we'll refer to two key concepts: **learnerObjects** and **digitalTwins**. DigitalTwins are nodes containing all information regarding objects we classify as digital twins, such as engines, aircraft, or ships. LearnerObjects, on the other hand, refer to resources like PDFs, images, audio files, and videos.

The challenge lies in taking sentence summaries and mapping the nodes we care about while filtering out irrelevant words. This is where a machine learning model becomes crucial. While pre-trained models for entity extraction exist, we chose to use a blank SpaCy model and employ transfer learning to fine-tune its Named Entity Recognition (NER) component. This approach allowed us to focus on the entities relevant to our domain and assign them custom labels.

To train the model, we started with three test PDFs and manually annotated 36 sentences, labeling the entities we deemed essential. The loss began at approximately **319.86**, reflecting the blank model's starting point. It steadily decreased to **~3.36 throughout the training**, indicating the model's ability to effectively learn and adapt to recognizing domain-specific entities.

Once the training was complete, the custom 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 extracted entities form the foundation for constructing nodes and relationships within the graph database.

Content Tracking

A significant change we made to the message body was the addition of a **contentID**. This identifier was introduced to track extracted components from a single piece of content. For instance, if images are extracted from an uploaded PDF, we need to associate those images with their source document. Similarly, for an audio file and its corresponding transcript both are tracked using the same contentID. This addition complements the existing **JOB ID** used for time tracking and sequencing logs.

Each file type—documents, audio, images, etc.—also has its own unique ID, such as **DocumentID**, **AudioID**, or **ImageID**, which are unique 256-bit hashes with timestamps. The **contentID**, however, serves a broader purpose, linking all related extracted components to their source content. In the

database, this means that a search for `contentID=2` will retrieve all associated media and metadata extracted from that specific content.

The contentID plays a crucial role in processing files and managing storage. Once content processing is complete, our system calls functions to retrieve messages matching the specified contentID. For example, when a document finishes processing in the Document Module, its outputs—such as `summary.txt`, `meta.txt`, and other files—are sent to the storage queue. Our function scans through the messages in the queue, matching them to the given contentID. Messages that don't match are skipped, and acknowledgments are not sent for those. The function halts once all relevant messages are found and processed.

Integration with the Image Module

We also utilized this contentID functionality in our integration with the Image Module. In this process, the contentID is passed to the module to retrieve images associated with a document. The module then classifies these images, ensuring they are linked to the original source content. This seamless tracking and integration enable accurate organization and retrieval of related data across the system.

Overall, the addition of the contentID has enhanced our ability to manage, track, and integrate data efficiently, supporting the scalability and robustness of the system.

Section 1 – What each code file does:

parse.py

The parse.py file serves as a server-side application designed to handle BSON-encoded data received over a TCP socket. It listens on a specified port, accepts client connections, and processes incoming BSON objects. The file decodes these objects, categorizes their content (e.g., documents, images, audio, and video), and publishes them to respective RabbitMQ queues based on their data type using topic-based routing. Additionally, it sends status updates to a dashboard queue, detailing the processing outcomes for each item. The file includes error handling to ensure robust communication and feedback in case of processing failures, logging issues, or missing data. This design facilitates efficient message processing and real-time status tracking in systems requiring content categorization and task distribution.

Changes made to parse.py:

BSON Encoding/Decoding: The OGparse.py uses bson.loads and bson.dumps, while the second uses BSON.encode and BSON.decode.

Socket Port: The first file binds to port 12345, while the second binds to port 12349.

Dashboard Messages: The second file adds a time field and detailed attributes (job_id, content_id, content_type), missing in the OGparse.py.

Queue Routing: The first file uses.Status. routing keys under the Topic exchange; the second routes to a Dashboard queue with an empty exchange. Check the Appendix for design decisions [1]

RabbitMQ Connection: The second file manages connections within parse_bson_obj for better efficiency, while the first re-establishes connections in each function.

Error Handling: The second file provides detailed error messages with timestamps in the Dashboard queue, whereas the first handles errors in publish_to_rabbitmq.

Routing Key Formatting: The second file dynamically formats content types by removing trailing "s" (data_type.rstrip('s')), which the first does not do.

Main_server_gui_sockets.py

The main_file_server_gui file provides a graphical interface for selecting, managing, and uploading various file types (documents, images, audio, and video) to a server. Users can dynamically add, remove, or clear files, with the application organizing the data into a job structure that includes unique IDs for logging and database storage. The selected files are sent to the server via a socket connection, with real-time status updates and notifications for success or errors. The file also integrates optional processing, such as an ImageClassifier for images, making it a comprehensive tool for file handling and server uploads.

Changes made to Main_server.py:

File Selection: The GUI file allows dynamic file selection using a graphical interface, while the second file hardcodes file paths and loads the data programmatically.

File Types and Validation: The GUI file uses filedialog for file selection with predefined filters for different file types (pdf, mp3, mp4, etc.). In contrast, the second file requires predefined paths and manually assigns file attributes such as DocumentType, AudioType, and VideoType.

Dynamic Updates: The GUI file dynamically updates the job dictionary as files are selected or removed. For example, it adjusts the count of documents, images, audio, and videos in real-time. The second file updates these counts manually in the hardcoded job dictionary.

File Uploading Workflow: The GUI file includes real-time status updates via status_label and uses messagebox to notify the user of success or failure during file uploads. The second file performs uploads silently without user feedback.

Error Handling: The GUI file handles interactive errors, showing alerts for missing file selections or upload issues. The second file relies on exceptions to handle errors programmatically without user notifications.

Unique ID Management: Both files use unique IDs for entities and jobs, but the GUI file explicitly links IDs like content_id for database storage and retains job_id for logging. The second file retains a more straightforward structure, focusing on ID, DocumentId, PictureID, AudioID, and VideoID.

Socket Port: The GUI file sends data to port 12349 (localhost), while the og file sends it to port 12345.

Modularity: The GUI file introduces modularity by separating file selection and upload workflows, enabling easy user interactions. The second file focuses on a straightforward programmatic execution.

Image Classifier Integration: The GUI file includes a commented integration for an ImageClassifier, which is designed to process images post-upload. This integration is currently inactive but will be utilized when the work on the image module progresses further. Specifically, for standalone image uploads (not extracted from documents), the ImageClassifier and its relevant import statement can be uncommented and activated. This would enable the file to call the image module function to process images directly from the queue using the provided content ID as a parameter. At present, the image module is invoked exclusively by the document module to classify images extracted from documents. These classified images are then stored with the corresponding learner object for structured storage and further processing.

```
if processed_job["NumberOfImages"] > 0:
    # Extract the content_id from the image
    image_content_id = processed_job["Images"][0]["content_id"]

    # Initialize ImageClassifier and pass the content ID
    processorImage = ImageClassifier()
    processorImage.consume_image(image_content_id)
```

React Frontend Uploader

We created a React app for uploading files, which is fully integrated with the dashboard. It simplifies file management by allowing users to upload and view documents, images, audio, and videos, while seamlessly communicating with the backend for processing and storage. This streamlined approach enhances usability and keeps the system efficient.

The FileUploader.js component provides a user interface for uploading multiple types of files (documents, images, audio, and videos). It manages file selection, display, and upload functionality while providing real-time status updates and error handling. The component uses React hooks for state management and integrates with Lucide icons for visual elements.

1. State Management:
    - files: Manages uploaded files state with separate tracking for:
        - document (single file)
        - image (single file)
        - audio (multiple files array)
        - video (single file)
    - status: Tracks upload process status
    - error: Handles error messages
2. handleFileSelect(type, event)
    - Purpose: Processes file selection for different file types
    - Key Steps:
        - Handles single file selection for document, image, and video
        - Manages multiple file selection for audio
        - Updates state while preserving existing files
3. removeFile(type, index)
    - Purpose: Removes selected files from the interface

- Key Steps:
    - Handles individual file removal for document, image, and video
    - Manages array-based removal for audio files
    - Updates state to reflect removals

4. handleUpload()
    - Purpose: Manages file upload process to backend
    - Key Steps:
        - Validates file selection
        - Creates FormData with selected files
        - Sends POST request to upload endpoint
        - Manages upload status and errors
        - Clears file selection after successful upload
5. FileSection Component
    - Purpose: Reusable component for file type sections
    - Features:
        - Customizable icon and file type acceptance
        - File display with remove capability
        - Upload button with file type filtering
        - Supports single and multiple file selections

UI Components:

1. File Type Sections:
    - Document (.pdf files)
    - Image (image files)
    - Audio (audio files, multiple)
    - Video (video files)
2. Status Indicators:
    - Error messages
    - Upload status
    - Upload progress

3. Upload Controls:
    - Main upload button
    - Individual file selection buttons
    - File removal buttons

The component integrates with the backend through a REST API endpoint /upload and provides real-time feedback on upload status and errors while maintaining a clean and intuitive user interface.

file_uploader_backend.py

file_uploader_backend.py is the FastAPI backend system for uploading files for the React Frontend Uploader above. It provides a REST API endpoint for handling file uploads from the frontend. It processes multiple file types (documents, images, audio, and video), creates structured job objects, and forwards them as BSON-encoded messages to a processing service.

Key Functions:

1. compute_unique_id(data_object)

    ● Purpose: Generates unique identifiers for jobs and files

    ● Key Steps:

    - Creates hash from BSON data, timestamp, and random value
    - Returns SHA-256 hash as hexadecimal string
2. send_bson_obj(job)

    ● Purpose: Sends BSON-encoded job to processing service

    ● Key Steps:

    - Establishes async connection to localhost:12349
    - Transmits encoded job data
    - Handles connection closure and errors
3. upload_files Endpoint (/upload)

    ● Purpose: Handles multipart file uploads and job creation

    ● Processes:

- Document (single file)
- Image (single file)
- Audio (multiple files)
- Video (single file)

- Job Structure Creation:

  - Assigns unique IDs to job and files
  - Includes file metadata and content
  - Tracks file counts and types
  - Maintains consistent ID references

File Processing Features:

1. File Handling:
   - Async file reading
   - Multiple file type support
   - File extension extraction
   - Binary content management
2. ID Management:
   - Job-level IDs
   - Content-specific IDs (DocumentId, PictureID, etc.)
   - Content tracking IDs
   - Consistent ID propagation

doc_module.py

The file listens for documents in the queue, processes one document at a time, and extracts its images, summary, metadata, and keywords. The extracted summary sentences are sent to the Analyzer.py module for entity and relationship extraction. The RecieverNParse module is then invoked to create a folder named after the content_id (store_contentID), where all related files are stored. If images are extracted from the document, they are added to the queue, and the image module is called to process them further, ensuring all document-related content is systematically organized and processed.

Changes made to document_module.py:

1. **Summary Format**:
    - **doc_module.py**: Generates the summary as an array of individual sentences for better structure and readability.
    - **document_module.py**: Concatenates selected sentences into a single string.
2. **Content ID Usage**:
    - **doc_module.py**: Introduces and uses content_id extensively for tracking and linking metadata, images, and relationships across modules.
    - **document_module.py**: Relies mainly on DocumentId for identification, without a dedicated content_id.
3. **Message Body Enhancements**:
    - **doc_module.py**: Adds detailed fields to image and status messages, such as content_id, PictureID, job_id, media_id, status, and message, providing richer metadata for each type of content.
    - **document_module.py**: Only fields like ID and Payload are included in messages.
4. **New Functions Added**:
    - **entity relation extraction.analyze(summary, file_name, contentID)**:
        - Integrated into the updated file to analyze text summaries, extract entities and relationships, and provide enriched metadata for downstream modules.
        - Absent in the document_module.py.
    - **remove_files() Updates**:
        - Enhanced in the updated file to remove temporary directories (e.g., images) and their contents after processing.
        - In the document_module.py, cleanup is limited to specific files without managing directories.
5. **Fields Added to Image Messages**:
    - The updated file adds content_id, PictureID, media_id, job_id, status, and message fields to the image message. These fields improve traceability, processing clarity, and downstream module integration.
    - The document_module.py only includes ID, PictureID, PictureType, FileName, and Payload for images.
6. **Enhanced Cleanup**:
    - **doc_module.py**: Cleans up all temporary files, directories, and images post-processing.
    - **document_module.py**: Limited cleanup functionality, focusing only on specific files.

7. **Image Handling Improvements**:
    - **doc_module.py**: Iterates through extracted images, links them with content_id, and adds metadata fields like job_id and media_id.
8. **Enhanced Publishing Logic**:
    - **doc_module.py**: Separates publishing logic for processing messages and status updates, ensuring payloads are excluded in status messages and providing detailed feedback.
    - **document_module.py**: Combines these steps.

analyzer.py

Analyzer.py file analyzes the document text summary using our fine-tuned spacy NLP NER to extract labels for the nodes we trained it to recognize; those extracted nodes get built and packaged with the appropriate information like the node type, digital twin Type (Aircraft, ships, engine, etc.) before it is sent to the DB the main node gets passed to the meta llama mission profile extractor to generate a mission profile statement. Once that is done. The package of nodes is sent to the dbOperationLocal.py to parse the array of nodes and relationships and store them into neo4j.

Global Initialization

1. **spacy Model Loading**:
    - A trained custom NER model (custom_ner_modelREL) is loaded into spacy.

      nlp = spacy.load(model_path)

Class: entityRelationExtraction

Method: analyze(sentences, file_name, contentID)

1. **Initialization**:
   Initializes an empty nodes list to store extracted entities, relationships, and additional metadata.

      Adds a learnerNode and learnerRelation as fixed entries:
      learnerNode = [file_name, "learnerObject", "pdf"]
      learnerRelation = ["learnerObject"]

2. **Entity Processing**:

   Iterates over the input sentences and processes each with the loaded spacy model to extract entities:

   ```
   for sentence in sentences:
     doc = nlp(sentence)
     for ent in doc.ents:
   ```

3. **Node Creation**:

   - For each entity (ent), creates a node with:

     - ent.text (the entity's text),
     - nodeType (digitalTwin), and
     - digitalTwinType (e.g., Aircraft or Engine).

```
nodeType = ent.label_[:11]
digitalTwinType = ent.label_[11:] #Splits the Label eg ent.Label = digitalTwinAircraft; digitalTwin is 11 character so we split it into node type and digitalTwinType AKA vehicle types, engines, generator etc
currentNode = [ent.text, nodeType, digitalTwinType]
```

Adds the created node to the nodes list:

```
nodes.append(currentNode)
```

4. **Entity Count Tracking**:
   Tracks the frequency of each entity in entity_counts for later determination of the **main topic**:
   ```
   normalized_entity = ent.text.strip().lower()
   ```

```
    entity_counts[normalized_entity] =
    entity_counts.get(normalized_entity, 0) + 1
```

5. **Relationship Extraction**:

Checks for relationships between the last two nodes in the nodes list using a predefined relationship_map:

```
prev_node = nodes[-2]
current_node = nodes[-1]
prev_type = prev_node[1] + prev_node[2]
curr_type = current_node[1] + current_node[2]
relationship = relationship_map.get((prev_type, curr_type))
```

If a valid relationship exists, insert it as a new node (relation_node) into the nodes list:

```
if relationship:
    relation_node = [relationship]
    nodes.insert(len(nodes) - 1, relation_node)
```

6. **Determine Main Topic**:

The most frequent entity in entity_counts is selected as the **main topic**:

```
if entity_counts:
    main_topic = max(entity_counts, key=entity_counts.get)
```

This takes advantage of explicit mention of the words but for implicit differentiation, the model would need to be trained with lots of data or a large pre-trained model.

7. **Duplicate Node Removal**:

The remove_duplicate_nodes function removes duplicates from the nodes list:

```
nodesUnique = remove_duplicate_nodes(nodes)
```

#This removes the number of times an entity is mentioned from the nodes list, just keeps 1 iteration

8. **Main Topic Node Extraction**:
   Finds the node corresponding to the **main topic** using find_main_topic_node:
   main_topic_node_copy = find_main_topic_node(nodesUnique, main_topic)

9. **Mission Profile Extraction**:

Creates a prompt based on the **main topic** and sends it to a Meta Llama 2:7b model hosted on a flask server with an API call to extract a mission profile, which is appended to the **main topic node**:

```
 prompt = f"What is the {mainNodeName} and what does it do? Two sentences max."
missionProfile = missionProfileExtraction(prompt)
main_topic_node_copy.append(missionProfile)
```

10. **Final Node List Construction**:

    - Adds the following items to the beginning of the nodesUnique list:
      - contentID: Unique content identifier.
      - learnerNode: Represents the file as a learner object.
      - learnerRelation: Describes the relationship type.
      - The main topic node (with the mission profile appended).

```
nodesUnique.insert(0, main_topic_node_copy)
nodesUnique.insert(0, learnerRelation)
nodesUnique.insert(0, learnerNode)
nodesUnique.insert(0, contentID)
```

11. **Parsing and Building**:

Sends the final nodesUnique list to the nodeBuilder.package parser function for processing and storage:

```
nodeBuilder.packageParser(nodesUnique)
```

The nodeBuilder class is in the dbOperationLocal.py file

Helper Functions

missionProfileExtraction(mainNode)

- **Purpose**: Sends a prompt to an external API to generate a mission profile for the main topic.
- **Steps**:
    1. Sends the mainNode as input to a local API endpoint (http://127.0.0.1:5002/generate).
    2. Extracts the generated_text field from the API response.

```
response = requests.post(url, json=payload)
data = response.json()
generated_text = data.get("generated_text", "No text generated.")
```

find_main_topic_node(nodesUnique, main_topic)

- **Purpose**: Locates the node corresponding to the **main topic** within the nodesUnique list.
- **Steps**:
    1. Iterates over the nodes to find one containing the main_topic in its text.
    2. Returns a deep copy of the matching node.

```
for node in nodesUnique:
    if main_topic.lower() in node[0].lower():
        main_topic_node_copy = copy.deepcopy(node)
        return main_topic_node_copy
```

remove_duplicate_nodes(nodes)

- **Purpose**: Removes duplicate nodes from the nodes list to ensure uniqueness.
- **Steps**:
    1. Iterates through nodes and appends only unique nodes to a new list.

```
unique_nodes = []
for node in nodes:
    if node not in unique_nodes:
        unique_nodes.append(node)
```

How the Nodes List is Built

1. **Entities**:
    - Extracted from sentences using spacy and stored in the nodes list with their types (nodeType and digitalTwinType).
2. **Relationships**:
    - Derived from pairs of consecutive nodes based on the relationship_map and inserted between the nodes.
3. **Learner Information**:
    - The learnerNode and learnerRelation are prepended to the list to represent the document.
4. **Mission Profile**:
    - A mission profile is generated for the **main topic** and appended to the corresponding node.
5. **Final Assembly**:
    - The contentID, learner information, and unique nodes are combined to form the final nodesUnique list.

Example Node List

Before Relationships:

```
[
    ["F18", "digitalTwin", "Aircraft"],
    ["GE414", "digitalTwin", "Engine"]
]
```

After Relationships:

```
[
    ["F18", "digitalTwin", "Aircraft"],
    ["engine"],
    ["GE414", "digitalTwin", "Engine"]
]
```

Final List: before the mission profile

```
[
    "contentID123",
    ["FileName.pdf", "learnerObject", "pdf"],
    ["learnerObject"],
    ["F18", "digitalTwin", "Aircraft", "Mission Profile Text"],
    ["engine"],
    ["GE414", "digitalTwin", "Engine"]
]
```

Example:

```
[
    "2d4b80013a941e55af042febd52deb4dd214f9ad87c96213cbf8d90d5a3dfed8",  # contentID
    ["F18GeEngineWithPic.pdf", "learnerObject", "pdf"],  # Learner node (file details)
    ["learnerObject"],  # Learner relation
    [
        "GE414 engine",
        "digitalTwin",
        "Engine",
        "The GE414 is a high-performance turbofan engine developed by General Electric (GE) for business jets and large aircraft. "        "It provides increased power, efficiency, and reliability compared to previous engines, resulting in lower operating costs and improved fuel consumption."
    ],
```

```
# Main topic node with the mission profile

    ["GE414 engine", "digitalTwin", "Engine"],  # Engine node
    ["engine"],  # Relationship node
    ["F18 aircraft", "digitalTwin", "Aircraft"]  # Aircraft node
]
```

dbOperationLocal.py

dbOperationLocal - The dbOperationsLocal file handles interactions with a Neo4j graph database, focusing on creating nodes, establishing relationships, and querying paths. It includes functions like getAllNodes for retrieving and inspecting nodes, nodeTraceback for tracing relationships starting from a specified learner object, and nodeTracebackManual for interactively exploring nodes and their connections. The file ensures consistency in relationship creation through functions like addLearnerRelation, addDigitalTwinRelation, and addImageLearner, which manage bidirectional relationships and attach properties like mission profiles and content IDs. Additionally, it integrates with a status feed to log and report on stored nodes and relationships.

1. getAllNodes()

- **Purpose**: Retrieves all nodes from the Neo4j database, limited to the first 100 nodes.
- **Key Steps**:
    1. Connects to Neo4j using the GraphDatabase driver.
    2. Runs a Cypher query (MATCH (n) RETURN n LIMIT 100) to fetch nodes.
    3. Iterates through the results and prints each node.

2. nodeTraceback(learnerObject, contentID)

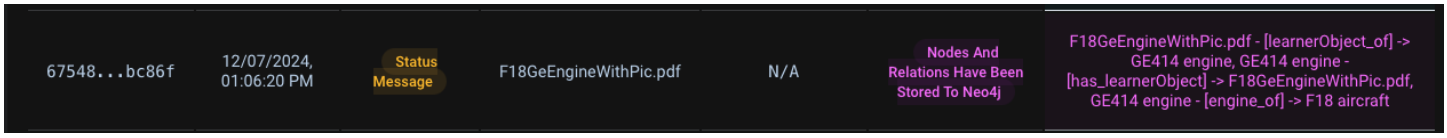
- **Purpose**: Traces relationships starting from a specific learnerObject node.
- **Key Steps**:
    1. Connects to the Neo4j database.
    2. Executes a Cypher query to match a path starting from the specified learnerObject and traversing relationships like learnerObject_of and others.

```
MATCH path = (startNode {name: $nodeToLookFor})-[:learnerObject_of]->(intermediateNode)-[nextRel]->(connectedNode)

RETURN startNode, relationships(path) as rels, nodes(path) as nodes
```

startNode: Starting node matched by name.
learnerObject_of: Traverses specific relationships to connected nodes.
Returns all nodes and relationships in the path.

3. Extracts and deduplicates relationships and nodes from the results.
4. Builds a status message using statusFeed and logs it.



3. nodeTracebackManual()

- **Purpose**: Allows manual selection of a node by partial name and traces all relationships connected to it.
- **Key Steps**:
    1. Asks the user for partial input to search for a node.
    2. Finds matching nodes with a Cypher query (MATCH (n) WHERE n.name CONTAINS $partialName).
    3. Prompts the user to select a node from the matching results.
    4. Deduplicates relationships and optionally uses the results for further processing.

```
MATCH path = (startNode {name: $nodeToLookFor})-[:learnerObject_of]->(intermediateNode)-[nextRel]->(connectedNode)

RETURN startNode, relationships(path) as rels, nodes(path) as nodes
```

4. addLearnerRelation(node1array, relation, node2array)

- **Purpose**: Adds a relationship between a learner object (e.g., a document or PDF) and another node in the graph.
- **Key Steps**:
    1. Extracts details from node1array (e.g., learnerObject, mediaType, contentID).
    2. Creates or merges nodes using MERGE Cypher statements and adds relationships (has_{relation} and {relation}_of).

```
MERGE (node1:`{learnerObject}`:`{mediaType}` {name: $nameofNode1})
MERGE (node2:`{primaryType2}`:`{secondaryType2}` {name: $nameofNode2})
MERGE (node1)<-[:`has_{relation}`]-(node2)
MERGE (node2)<-[:`{relation}_of`]-(node1)
```

Purpose: Links a learner object to another node.

- node1: A learner object node is created/ensured.
- node2: A related node is created/ensured.
- MERGE: Ensures bidirectional relationships.

3. Inserts a mission profile (missionProfile2) if the second node is newly created.

5. addDigitalTwinRelation(node1array, relation, node2array)

- **Purpose**: Adds a relationship between two digital twin nodes in the graph.
- **Key Steps**:
    1. Extracts node types and attributes from node1array and node2array.
    2. Creates or merges the nodes and adds bidirectional relationships (has_{relation} and {relation}_of) between them.

```
        MERGE (node1:`{primaryType1}`:`{secondaryType1}` {name:
$nameofNode1}) MERGE (node2:`{primaryType2}`:`{secondaryType2}` {name:
$nameofNode2}) MERGE (node1)<-[:`has_{relation}`]-(node2) MERGE
(node2)<-[:`{relation}_of`]-(node1)
```

Purpose: Connect two digital twin nodes.

Key Points:

- Both nodes (node1 and node2) are typed as digital twins with primary and secondary types.

- Relationships ensure bidirectional connections.

6. addImageLearner(node1array, relation, mainContentID)

- **Purpose**: Creates a node for an image and links it to a main content node (e.g., a document).
- **Key Steps**:
    1. Extracts image-related details (predictedClass, contentID, etc.) from node1array.
    2. Creates the image node with properties like name, location, and predictedClass.
    3. Links the image node to the main learner node using bidirectional relationships, using contentID to find the originating learnerObject.
    4. Sends a status message to the statusFeed module to confirm the image was stored and classified.

```
CREATE (node1:`{learnerObject}`:`{mediaType}` {name: $nameofNode1, contentID: $contentID, location: $location}) MERGE (node2 {contentID: $mainContentID}) CREATE (node1)<-[:`has_{relation}`]-(node2) CREATE (node2)<-[:`{relation}_of`]-(node1)
```

7. imagePackageParser(package, contentID)

- **Purpose**: Processes a package of image nodes and relationships for storage.
- **Key Steps**:
    1. Loops through the package until all items are processed.
    2. Extracts each node, processes it with addImageLearner, and deletes it from the package.

8. packageParser(package)

- **Purpose**: Processes a package of nodes and relationships for a learner object and its related digital twins and calls functions to store them.

9. statusFeed.messageBuilder

- **Purpose**: Sends a status message to the system's status feed. Used in:
    - nodeTraceback for storing relationships.
    - addImageLearner for image classification updates.

Step-by-Step:

1. **Extract contentID**:
    - contentID = "contentID123"
    - Remove it from the package.
2. **Add Learner Relation**:
    - node1array = ["F18GeEngineWithPic.pdf", "learnerObject", "pdf", "contentID123"]
    - relation = "learnerObject"
    - node2array = ["GE414 engine", "digitalTwin", "Engine"]
    - Calls addLearnerRelation(node1array, relation, node2array).
3. **Process Remaining Digital Twin Relationships**:
    - Loops through the remaining package in chunks of 3:
        - First iteration:
            - node1array = ["GE414 engine", "digitalTwin", "Engine"]
            - relation = "engine"
            - node2array = ["F18 aircraft", "digitalTwin", "Aircraft"]
            - Calls addDigitalTwinRelation(node1array, relation, node2array).
4. **Traceback**:
    - Calls nodeTraceback("F18GeEngineWithPic.pdf", "contentID123") to validate, log relationships and send message to dashboard status

statusFeed.py

The statusFeed.py file is responsible for building message/job bodies. It works with the publisher.py file to send job objects off to RabbitMQ.

1. **messageBuilder(content_ID)**
    - Creates the job body (a dictionary) which consists of the jobID, contentID, status, timestamp, and details.
    - Calls `messageSender` with the newly made job as a parameter.
2. **messageSender(bsonObj)**
    - Calls `publish_to_rabbitmq` to send the job off to publisher.py
    - Does this using: `from publisher import publish_to_rabbitmq`

publisher.py

The publisher.py file takes the job object made by statusFeed.py and converts it into the BSON format. It then sends the message off to RabbitMQ using the specified port. Run in console before running the statusFeed.py file.

1. **recvall(sock, expected_length)**
    - Ensures all data is received over a channel.
    - Continuously reads until entire length is received.
    - Raises exception if channel closes early
2. **handle_client(client)**
    - Reads length of BSON object, then the rest of the BSON data with `recvall`
    - Turns BSON data into python object, then passes it through `parse_bson_obj`
3. **parse_bson_obj(obj)**
    - Maps data types to corresponding routing keys.
    - Iterates and sends to `publish_to_rabbitmq` according to data type.
4. **receive_bson_obj()**
    - Opens and keeps a connection open to listen out for incoming BSON objects.
5. **publish_to_rabbitmq(routing_key, message)**
    - Opens a channel for send off to RabbitMQ
    - Makes a copy of the message parameter and converts it to the BSON format.
    - Publishes the main message to port, and status message to dashboard.
    - Exception handling for failed publishing.
    - When finished, it closes the channel to RabbitMQ.

recNparse.py

The recNParse.py file handles the consumption and storage of messages from RabbitMQ queues, focusing on two main content types: store and image. Its MessageProcessor class establishes connections to RabbitMQ, declares necessary queues (Store, Document, and Image), and binds them to a Topic exchange.

1. create_dir(directory)

- Purpose: Creates a directory if it doesn't already exist.
- Key Steps:

- Checks if the directory exists using os.path.exists
- Creates directory using os.makedirs if it doesn't exist
- Logs creation of new directory
- Returns the directory path

2. save_file(file_path, data)

- Purpose: Saves binary data to a specified file path.
- Key Steps:
  - Creates necessary parent directories
  - Opens file in binary write mode
  - Writes data to file
  - Logs success or failure of file save operation
  - Includes error handling for file operations

3. process_store(ch, method, properties, body, content_id)

- Purpose: Processes store-type messages containing document data and metadata.
- Key Steps:
  - Decodes BSON message body
  - Verifies content ID matches
  - Creates storage directory structure
  - Saves payload as PDF
  - Extracts and saves Meta, Summary, and Keywords
  - Implements message acknowledgment
  - Handles errors with appropriate requeue strategy

4. process_image(ch, method, properties, body, content_id)

- Purpose: Processes image messages and saves associated files.
- Key Steps:
  - Decodes BSON message
  - Validates content ID match
  - Creates/uses target folder named "store_<contentID>"
  - Saves image with formatted filename
  - Acknowledges successful message processing
  - Handles errors with logging and requeue options

5. consume_image(contentID)

- Purpose: Consumes messages from the Image queue for a specific content ID.
- Key Steps:
  - Implements non-blocking message retrieval
  - Tracks delivery tags to prevent duplicates
  - Processes messages using process_image
  - Stops consumption when the queue has iterated through all the messages once or its empty
  - Includes error handling and logging

6. **consume_store(content_id)**

- Purpose: Consumes messages from the Store queue using callbacks.
- Key Steps:
  - Sets up callback for message processing
  - Monitors queue state
  - Stops consumption when the queue has iterated through all the messages once or its empty
  - Processes messages using process_store
  - Includes queue state monitoring
  - Implements proper channel management

Each function works together to ensure reliable message processing and file storage while maintaining proper error handling and logging throughout its operations. The module integrates with the broader system to provide a complete document and image processing pipeline.

Image_module.py

Function number 5 above integrates with the image module by identifying images with associated `contentID`. It then calls the `classify_image` function, which classifies all images, saves the resulting files, builds corresponding nodes, and appends these nodes to a list. Finally, it calls the `imagePackage` parser to store the data in Neo4j.

```
                predictedString = (f"{predicted_class}, Confidence:
{confidence_score:.2f}")
                imageLearner = [
                    file_name,
                    "learnerObject",
                    "Image",
                    target_folder,
                    content_id,
                    predictedString,
                ]
                imageNodes.append(imageLearner)

            except Exception as e:
                logging.error(f"Error in classification: {e}")
                raise

            # Save the file in the target folder
            image_file_name = f"image_{file_name}+{message_content_id}.png"
            self.save_file(os.path.join(target_folder, image_file_name),
image_data)

            # Add classification results to log message
            logging.info(
                f"Processed image with Content ID: {message_content_id},
Class: {predicted_class}"
            )
            ch.basic_ack(delivery_tag=method.delivery_tag)
            # functionCall here to send to DB
```

```
        nodeBuilder.imagePackageParser(imageNodes, message_content_id)
```

Section 2 – How To Setup:

- https://hackmd.io/8EaD1LutTx2O1CCeuQwKmQ?both

Section 3 – How To Use Neo4j:

https://www.youtube.com/watch?v=c_hldeLPN0g&ab_channel=MohammadChowdhury

Section 4 – How To Train SpaCy Model:

SpacyTraining.py

This section focuses on how to train a SpaCy model to familiarize it with a dataset and enable it to recognize specific entities within text. The training process is outlined in the `SpacyTraining.py` file, which is designed to build a Named Entity Recognition (NER) model capable of identifying entities like aircraft, engines, and tanks and categorizing them under labels such as “digitalTwinEngine” or “digitalTwinAircraft.” The model is trained with annotated text data, and its performance improves with more iterations due to fine-tuning.

The process begins by initializing a blank English language model using `nlp = spacy.blank("en")`. This step sets up a model from scratch, which is suitable for custom training. The code checks whether the NER component exists in the pipeline using `if "ner" not in nlp.pipe_names`. If it does not, the NER component is added to the pipeline.

The training data is defined in the `TRAIN_DATA` section, consisting of pairs of text and corresponding entity annotations. Each annotation specifies the start and end positions of entities within the text, along with their labels. This structured format allows the model to learn how to identify and categorize these entities. More data can be made with dataTrainer.py

The NER component is then populated with the entity labels extracted from the annotations in `TRAIN_DATA`. This is achieved using a loop: `for _, annotations in TRAIN_DATA`, which adds the labels to the NER model, enabling it to associate entities with their respective categories.

To ensure efficient training, the `Unaffected_pipes` feature disables all pipeline components except the NER during training. By using `with nlp.disable_pipes(*unaffected_pipes):`, this ensures that only the NER component is updated, avoiding interference from other pipeline components like the parser or tagger.

The training is performed using `nlp.update(examples, drop=0.5, losses=losses)`. This function processes batches of examples, updating the model with each iteration. A dropout rate is applied to improve generalization, and the losses are recorded to monitor the training progress. Increasing the number of iterations enhances the model's accuracy by gradually refining its predictions.

Finally, the trained model is saved to disk with `nlp.to_disk("./custom_ner_modelREL")`. This step allows the model to be reused or further fine-tuned. Saving the model ensures that the training process does not need to be repeated from scratch, making it accessible for practical applications.

`SpacyTraining.py` provides a structured approach to training a custom NER model with spaCy, from initializing a blank model to saving the trained version for future use. The model becomes adept at recognizing and categorizing entities within text by iterating over annotated data and fine-tuning the NER component.

dataTrainer.py

The **dataTrainer.py** file is designed to process sentences and identify phrases corresponding to labeled entities, such as "digitalTwinEngine," using regular expressions. This utility we built generates training data for the Named Entity Recognition (NER) model by pairing each sentence with the entities it contains and their respective labels. This file enables the preparation of data required to train the spaCy model to recognize and categorize entities effectively.

A list of sentences, defined in the sentences variable, serves as the input text to be analyzed for entity recognition. Each sentence may include terms that reference specific entities.

The file also defines a dictionary called phrases_to_index, which maps key phrases to their respective entity labels. For instance, the "GE414 engine" might be associated with the label "digitalTwinEngine." This dictionary acts as a reference for the entity-label relationships.

An empty list, training_data, is initialized to store the processed sentences paired with their identified entities. The file iterates through each sentence in the sentences list. Within this loop, a nested loop compares each phrase from the phrases_to_index dictionary to the current sentence using regular expressions. The re.search function ensures that exact matches of phrases are identified, with the use of `\b` for word boundaries to avoid partial matches and re.escape to handle special characters.

When a match is found, the start and end positions of the phrase in the sentence are calculated, and these positions, along with the corresponding entity label, are stored in the entities list. This process captures the necessary details to associate the identified phrase with its label and its location within the text.

Once all entities in a sentence are identified, the file appends the sentence along with its entity annotations to the training_data list. Each entry in this list contains the original sentence and a dictionary specifying the entities, their positions, and their labels. Finally, the file prints the processed training_data to display the results, providing a clear view of the sentences and their labeled entities that will be used to train the NER model.

This structured approach ensures that the training data is well-organized and ready for use in building a spaCy model capable of recognizing and classifying entities accurately.

Examples of sentences and phrases and the dataset itself:

```
('MG78 turbine section.', {'entities': []}),
('MG78 turbine section.', {'entities': []}),
('MG78 turbine section.', {'entities': []}),
('The GE414 engine powers the F18 aircraft.', {'entities': [(4, 16, 'digitalTwinEngine'), (28, 40, 'digitalTwinAircraft')]}),
('The GE54 engine powers the F16 aircraft.', {'entities': [(4, 15, 'digitalTwinEngine'), (27, 39, 'digitalTwinAircraft')]}),
('The F16 aircraft is equipped with a GE401 engine.', {'entities': [(4, 16, 'digitalTwinAircraft'), (36, 48, 'digitalTwinEngine')]}),
('The F18 aircraft requires regular maintenance for its GE534 engine.', {'entities': [(4, 16, 'digitalTwinAircraft'), (54, 66, 'digitalTwinEngine')]}),
('The M761 tank uses a Titan65 engine.', {'entities': [(4, 13, 'digitalTwinGround'), (21, 35, 'digitalTwinEngine')]}),
('The Titan65 engine is a high-performance model.', {'entities': [(4, 18, 'digitalTwinEngine')]}),
('The M761 tank uses a Titan65 engine.', {'entities': [(4, 13, 'digitalTwinGround'), (21, 35, 'digitalTwinEngine')]}),
('The F22 aircraft features the advanced GE556 engine.', {'entities': [(4, 16, 'digitalTwinAircraft'), (39, 51, 'digitalTwinEngine')]}),
('Regular maintenance is required for the Titan65 engine and the M761 tank.', {'entities': [(63, 72, 'digitalTwinGround'), (40, 54, 'digitalTwinEngine')]}),
```

Above are some of the sentences and annotations of their locations in the newSpacyTraining.py file, explained previously above, and what we used to train the model explicitly.

Appendix

[1] Parse.py - The design choice to use an empty exchange for the Dashboard queue can improve efficiency in scenarios where messages are meant for a single, dedicated queue (like a Dashboard). This simplifies routing and reduces the overhead of pattern matching or topic-based routing, making it more straightforward to deliver high-volume status updates. For example, if a large number of messages are being sent exclusively to a dashboard, an empty exchange allows for direct delivery without the need for routing key evaluation.

Meanwhile, .Status. under the Topic exchange is retained for flexibility where messages might need to be routed to multiple consumers or queues based on patterns. This hybrid approach balances efficiency with scalability, allowing the system to handle both dedicated and generalized routing needs.