Design Document: Automated Offloading of NER and Fuzzy Matching Processing to HPC Cluster

Table of Contents

1. Introduction
2. Objectives
3. System Architecture
4. Components Description
5. Data Flow
6. Security Considerations
7. API Design
8. Error Handling and Data Consistency
9. Integration with Existing Scripts
10. Deployment and Infrastructure
11. Automation
12. Monitoring and Logging
13. Maintenance and Scalability
14. Conclusion
15. Appendices

---

1. Introduction

This design document outlines the implementation of an automated system to offload intensive Named Entity Recognition (NER) and fuzzy matching processing tasks from a local MongoDB/Flask application to a High-Performance Computing (HPC) cluster. Leveraging the HPC's computational power will significantly enhance processing efficiency, especially as the dataset scales beyond 200k documents. The goal is to maintain data integrity, ensure security, and achieve seamless integration between the local server and the HPC cluster.

---

2. Objectives

- **Automation:** Fully automate the workflow for identifying, processing, and integrating data between the local server and HPC cluster.
- **Security:** Ensure secure data transmission and restrict access to authorized entities.
- **Data Consistency:** Maintain synchronization and consistency between local MongoDB documents and processed results.

- **Efficiency:** Optimize data transfer by handling only necessary updates using delta updates and efficient data formats.
- **Reliability:** Implement robust error handling and idempotent operations to prevent data corruption and ensure reliable processing.
- **Scalability:** Design the system to accommodate future offloaded calculations beyond NER and fuzzy matching.

---

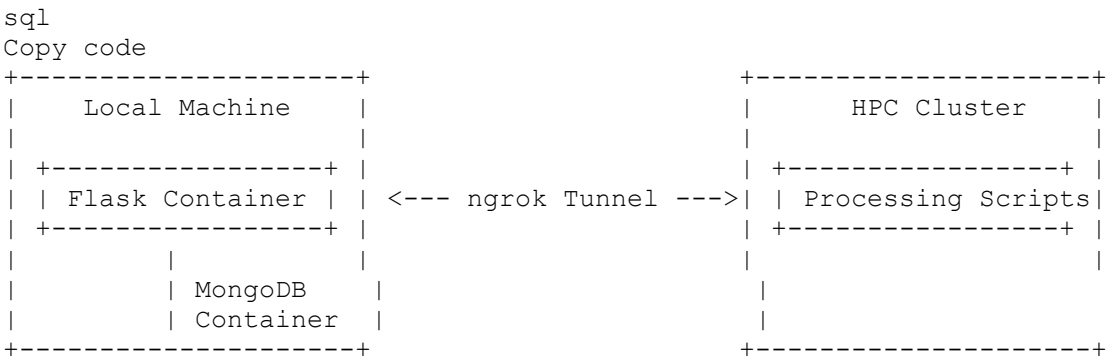
3. System Architecture

3.1 Overview

The system comprises two main environments:

1. **Local Environment:**
    - **Flask Application Container:** Hosts the web interface and manages the local MongoDB as the authoritative data source.
    - **MongoDB Container:** Stores the definitive dataset with a persistent data volume.
    - **ngrok Tunnel:** Securely exposes the Flask API to the HPC cluster.
2. **HPC Cluster:**
    - **Processing Scripts:** Execute NER and fuzzy matching tasks by interacting with the local server's API.
    - **Secure Connection:** Initiates connections to the local server via the ngrok tunnel.
    - **Extensible Processing Modules:** Facilitates the addition of future processing tasks.

3.2 Architectural Diagram



---

4. Components Description

4.1 Local Environment

- **Flask Application Container:**
  - Hosts the Flask web application.
  - Exposes RESTful API endpoints for data retrieval and update.
  - Interfaces with the local MongoDB container.
  - Implements health checks to ensure readiness.
- **MongoDB Container:**
  - Stores all documents as the authoritative data source.
  - Contains processed fields such as NER tags and fuzzy match scores.
  - Utilizes a persistent data volume to ensure data durability.
  - Executes initialization scripts for setting up collections and indexes.
- **ngrok Tunnel:**
  - Exposes the Flask API securely over the internet.
  - Provides HTTPS encryption for data in transit.
  - Implements authentication mechanisms to restrict access.
  - Configured with persistent URLs for consistent access.

4.2 HPC Cluster

- **Processing Scripts:**
  - Scripts responsible for performing NER and fuzzy matching tasks.
  - Utilize existing `data_processing.py`, `entity_linking.py`, and `ner_processing.py` for processing logic.
  - Designed to be modular to allow future offloaded calculations.
  - Containerized (e.g., Docker, Singularity) to ensure environment consistency.
- **Secure Connection:**
  - Initiates outbound connections to the local server via the ngrok tunnel.
  - Retrieves data needing processing and sends back processed results.
- **Extensible Processing Modules:**
  - Structured to add new processing tasks with minimal changes to existing code.
  - Encapsulates processing logic to maintain separation of concerns.

---

5. Data Flow

1. **Initialization:**
   - The local server starts and establishes an ngrok tunnel to expose its API securely.
2. **Task Identification:**
   - The HPC cluster connects to the local server's API via the ngrok tunnel.
   - It queries the API to retrieve a list of documents that require NER and fuzzy matching processing.
3. **Data Retrieval:**
   - The HPC pulls the necessary document data (e.g., document IDs and text content) using the API.
4. **Processing:**

    - The HPC performs NER and fuzzy matching on the retrieved documents using the existing processing scripts (`data_processing.py`, `entity_linking.py`, `ner_processing.py`).
5. **Result Transmission:**
    - The HPC sends the processed results back to the local server via a secure API endpoint.
    - Data is compressed and formatted efficiently (e.g., JSON or BSON) before transmission.
6. **Data Integration:**
    - The local server receives the processed data and updates the corresponding documents in MongoDB.
    - Ensures data consistency and handles idempotent operations.
7. **Confirmation:**
    - The local server acknowledges successful integration, and the processed documents are updated accordingly.

---

6. Security Considerations

6.1 Secure Tunneling

- **HTTPS Encryption:**
    - Use ngrok's HTTPS feature to encrypt all data transmitted between the HPC and the local server.
- **Authentication:**
    - Leverage ngrok's built-in authentication (e.g., basic auth) to ensure only authorized entities can access the API endpoints.
    - Implement API keys or OAuth tokens for additional security on the Flask API.

6.2 Firewall and Network Policies

- **Port Restrictions:**
    - Only expose necessary ports (e.g., the port running the Flask API) via ngrok.
    - Ensure that the local firewall blocks all other inbound connections.
- **IP Whitelisting:**
    - If possible, restrict API access to the HPC's IP addresses using ngrok's IP filtering features.

6.3 Data Protection

- **Data Encryption:**
    - Ensure data at rest in MongoDB is encrypted, especially sensitive information.
- **Access Controls:**

    - Implement role-based access controls (RBAC) in MongoDB to restrict who can read/write data.

6.4 Monitoring and Logging

- **Access Logs:**
    - Monitor ngrok and Flask application logs to detect any unauthorized access attempts.
- **Audit Trails:**
    - Maintain logs of all data transfers and processing activities for accountability and compliance.
- **Log Rotation:**
    - Configure Docker logging drivers with rotation policies to manage log file sizes.

---

7. API Design

7.1 Overview

The Flask application will expose the following API endpoints to facilitate communication with the HPC cluster:

1. **GET /api/processing-tasks**
    - **Purpose:** Retrieve a list of documents that require processing.
    - **Response:** JSON array containing document IDs and necessary fields for processing.
2. **POST /api/processed-data**
    - **Purpose:** Receive processed NER and fuzzy matching results from the HPC.
    - **Request Body:** JSON array containing document IDs and their corresponding processed data.
    - **Response:** Confirmation of successful data integration.

7.2 Endpoint Specifications

7.2.1 GET /api/processing-tasks

- **Authentication:**
    - Requires a valid API key or OAuth token.
- **Parameters:**
    - `batch_size` (optional): Number of documents to retrieve per request.
- **Response Structure:**

```
json
Copy code
{
```

```
  "tasks": [
    {
      "document_id": "unique_id_1",
      "content": "Document text for processing."
    },
    {
      "document_id": "unique_id_2",
      "content": "Another document text."
    }
    // ... more documents
  ]
}
```

7.2.2 POST /api/processed-data

- **Authentication:**
  - Requires a valid API key or OAuth token.
- **Request Body Structure:**

```
json
Copy code
{
  "processed_data": [
    {
      "document_id": "unique_id_1",
      "ner_tags": ["Entity1", "Entity2"],
      "fuzzy_match_scores": {"EntityA": 0.95, "EntityB": 0.89}
    },
    {
      "document_id": "unique_id_2",
      "ner_tags": ["Entity3"],
      "fuzzy_match_scores": {"EntityC": 0.92}
    }
    // ... more processed results
  ]
}
```

- **Response Structure:**

```
json
Copy code
{
  "status": "success",
  "updated_documents": ["unique_id_1", "unique_id_2"]
}
```

7.3 Data Formats and Compression

- **Data Formats:**
  - Use JSON for structured and lightweight data transfer.
  - Alternatively, use BSON for binary-encoded serialization if performance gains are necessary.
- **Compression:**

    - Compress the request payloads using gzip before transmission.
    - Implement compression/decompression on both the client (HPC) and server (Flask) sides.

---

8. Error Handling and Data Consistency

8.1 Idempotent Operations

- **Design:**
    - Ensure that processing tasks can be safely retried without causing duplicate updates or data corruption.
    - Use document IDs to check if processed data already exists before applying updates.
- **Implementation:**
    - On receiving processed data, verify if the `ner_tags` and `fuzzy_match_scores` fields are already populated.
    - If they are, decide whether to overwrite based on timestamps or other criteria.

8.2 Robust Error Handling

- **API Responses:**
    - Use appropriate HTTP status codes to indicate success or specific failure reasons.
    - Provide meaningful error messages in the response body for debugging.
- **Retry Mechanisms:**
    - Implement retries on the HPC side for transient failures such as network issues.
    - Use exponential backoff strategies to avoid overwhelming the server.

8.3 Data Validation

- **Incoming Data:**
    - Validate the structure and content of incoming processed data before integration.
    - Ensure that all required fields are present and correctly formatted.
- **Database Transactions:**
    - Utilize MongoDB's transaction capabilities to perform atomic updates.
    - Roll back transactions in case of failures to maintain data integrity.

8.4 Conflict Resolution

- **Versioning:**
    - Implement versioning for documents to handle concurrent updates gracefully.
    - Use MongoDB's `findAndModify` operations with version checks.
- **Audit Logs:**
    - Maintain logs of all updates to track changes and resolve conflicts if they arise.

---

9. Integration with Existing Scripts

9.1 Overview

The existing scripts (`data_processing.py`, `database_setup.py`, `entity_linking.py`, `ner_processing.py`) handle data ingestion, validation, and entity linking within the local environment. To integrate these scripts into the new architecture, modifications and extensions are necessary to facilitate remote invocation and data exchange with the HPC cluster.

9.2 Modifications and Enhancements

9.2.1 data_processing.py

- **Purpose:** Processes JSON and TXT files, inserts them into MongoDB, and updates field structures.
- **Enhancements:**
    - **API Integration:** Modify the script to accept data retrieved via the API instead of directly accessing the file system.
    - **Incremental Updates:** Ensure that only new or changed documents are processed and inserted.
    - **Remote Invocation:** Allow the script to be invoked remotely by the HPC cluster, possibly via command-line arguments or environment variables.

9.2.2 database_setup.py

- **Purpose:** Sets up MongoDB collections, indexes, and provides utility functions for database interactions.
- **Enhancements:**
    - **API Endpoint Expansion:** Ensure that the Flask API can utilize existing database functions for updating documents based on processed data.
    - **Security:** Implement additional checks to ensure that only authenticated requests can perform database modifications.

9.2.3 entity_linking.py

- **Purpose:** Performs entity linking using NER and fuzzy matching, integrating results back into MongoDB.
- **Enhancements:**
    - **Remote Execution:** Allow the script to accept data from the local server via API endpoints.
    - **Result Packaging:** Ensure that processed data is packaged in the required format for seamless transmission back to the local server.

9.2.4 ner_processing.py

- **Purpose:** Handles NER processing on documents, extracting entities and preparing them for linking.
- **Enhancements:**
  - **API Integration:** Modify the script to accept data via API requests and return results accordingly.
  - **Modular Processing:** Ensure that NER processing can be modularly invoked by the HPC scripts.

9.3 Workflow Integration

1. **HPC Processing Scripts:**
   - Utilize the existing processing logic from `data_processing.py`, `entity_linking.py`, and `ner_processing.py`.
   - Implement remote invocation capabilities to interact with the local Flask API for data retrieval and result submission.
2. **Local Flask API:**
   - Extend API endpoints to support receiving data from HPC scripts.
   - Integrate existing database utility functions to handle data updates based on processed results.
3. **Data Exchange:**
   - Ensure that the data formats between the local server and HPC cluster are compatible.
   - Utilize efficient serialization (e.g., JSON, BSON) and compression (e.g., gzip) for data transfer.

---

10. Deployment and Infrastructure

10.1 Local Environment Setup

- **Containers:**
  - **MongoDB Container:**
    - Persistent data volume to ensure data durability.
    - Expose necessary ports internally within the Docker network.
    - Includes health checks to verify readiness.
    - Mounts initialization scripts for setting up collections and indexes.
  - **Flask Application Container:**
    - Linked to the MongoDB container via a user-defined bridge network (`app_network`).
    - Configured to communicate with MongoDB using environment variables.
    - Exposes API endpoints required for processing tasks.
    - Utilizes a multi-stage Docker build for optimized image size and pre-installed dependencies.

        - Implements log rotation to manage log file sizes.
- **ngrok Configuration:**
    - Run ngrok on the host machine to expose the Flask API.
    - Use ngrok's configuration file to set up authentication and specify the local port.
    - Configured with persistent URLs for consistent access.
    - Secured with API keys or OAuth tokens to restrict access.

10.2 HPC Cluster Setup

- **Environment:**
    - Ensure that the HPC cluster has outbound internet access to connect to the ngrok tunnel.
    - Install necessary dependencies for running processing scripts (e.g., Python, NER libraries, RapidFuzz).
    - Utilize containerization (e.g., Docker, Singularity) for consistency with the local environment.
- **Processing Scripts:**
    - Develop or modify scripts to:
        1. Connect to the Flask API via ngrok.
        2. Retrieve documents needing processing.
        3. Perform NER and fuzzy matching using `data_processing.py`, `entity_linking.py`, and `ner_processing.py`.
        4. Compress and send processed data back to the local server.
    - Design scripts to be modular, allowing easy integration of additional processing tasks in the future.

10.3 Networking

- **ngrok Tunnel:**
    - Configured with persistent URLs or reserved domains to maintain consistent endpoints.
    - Ensures that the tunnel remains active when processing tasks are expected.
- **DNS and Domain Management:**
    - If using custom domains with ngrok, manage DNS settings accordingly to ensure seamless access from the HPC cluster.

---

11. Automation

11.1 Scheduling Processing Tasks

- **Trigger Mechanism:**
    - Set up a scheduled job on the HPC cluster (e.g., cron job) to initiate processing at defined intervals or based on specific triggers.

- **On-Demand Processing:**
    - Allow manual initiation of processing tasks via scripts if necessary.

11.2 Data Transfer Automation

- **Scripts:**
    - Develop automated scripts on the HPC to handle:
        - Authentication with the Flask API.
        - Retrieval of processing tasks.
        - Data compression.
        - Transmission of processed data back to the local server.
- **Workflow Orchestration:**
    - Use workflow managers (e.g., Apache Airflow) if the processing involves multiple dependent steps.

11.3 Integration with Containers

- **Container Management:**
    - Use Docker Compose or Kubernetes for managing containers in the local environment.
    - Ensure that containers restart automatically in case of failures.
- **Script Integration:**
    - Containerize processing scripts to ensure environment consistency between local and HPC deployments.
- **Extensibility:**
    - Design processing containers to accept different processing modules, facilitating the addition of new tasks without major changes.

---

12. Monitoring and Logging

12.1 Monitoring

- **Local Server:**
    - Monitor API performance and uptime using tools like Prometheus and Grafana.
    - Track MongoDB performance metrics to ensure database health.
- **HPC Cluster:**
    - Monitor processing script performance, resource utilization, and task completion rates.

12.2 Logging

- **API Logs:**

    - Implement comprehensive logging in the Flask application to record incoming requests, processing outcomes, and errors.
- **Processing Logs:**
    - Maintain logs on the HPC cluster for task retrieval, processing steps, and data transmission.
- **Centralized Logging:**
    - Use centralized logging solutions (e.g., ELK Stack) to aggregate and analyze logs from both environments.
- **Log Rotation:**
    - Configure Docker logging drivers with rotation policies to manage log file sizes.

12.3 Alerting

- **Set Up Alerts:**
    - Configure alerts for critical failures, such as:
        - Failed data transmissions.
        - API authentication errors.
        - Unusually high processing times.
- **Notification Channels:**
    - Use email, Slack, or other communication tools to receive alerts in real-time.

---

13. Maintenance and Scalability

13.1 Maintenance

- **Regular Updates:**
    - Keep all software components, including containers and dependencies, up to date with security patches.
- **Backup Strategies:**
    - Implement regular backups of the MongoDB data volume to prevent data loss.
    - Store backups securely, possibly in an offsite location or cloud storage.
- **Routine Checks:**
    - Periodically verify the integrity of the ngrok tunnel and ensure it is functioning as expected.

13.2 Scalability

- **Data Volume Growth:**
    - Design the system to handle increasing numbers of documents by:
        - Optimizing MongoDB indexes.
        - Scaling the HPC processing capacity as needed.
- **Parallel Processing:**

    - Enable the HPC cluster to process multiple tasks in parallel to reduce processing time.
- **Load Balancing:**
    - If necessary, implement load balancing strategies to distribute processing workloads efficiently.

13.3 Future Enhancements

- **Extended Processing Capabilities:**
    - Incorporate additional processing tasks beyond NER and fuzzy matching as needed.
- **Advanced Security Measures:**
    - Implement multi-factor authentication (MFA) for API access.
    - Utilize more sophisticated encryption standards for data at rest and in transit.
- **User Management:**
    - Develop role-based access controls within the Flask application to manage user permissions.
- **Modular Processing Framework:**
    - Develop a plugin-based processing framework to easily add new processing modules without altering the core system.

---

14. Conclusion

This design document presents a comprehensive plan to automate the offloading of NER and fuzzy matching processing tasks from a local MongoDB/Flask application to an HPC cluster. By leveraging secure tunneling with ngrok, implementing robust API authentication, and ensuring data consistency through idempotent operations and transactional updates, the system aims to achieve efficient, secure, and reliable processing. Additionally, by optimizing data transfer with delta updates and efficient data formats, the system minimizes bandwidth usage and enhances performance. Proper error handling, monitoring, and logging further ensure the system's resilience and maintainability, while the modular and encapsulated design facilitates future scalability and the addition of new processing tasks.

Appendices

A. Technology Stack

- **Local Environment:**
    - **Flask:** Web framework for the API and web interface.
    - **MongoDB:** NoSQL database for storing documents.
    - **Docker:** Containerization platform for Flask and MongoDB.

    - **ngrok:** Secure tunneling service to expose the Flask API.
- **HPC Cluster:**
    - **Processing Scripts:** Python scripts utilizing NER libraries (e.g., spaCy), fuzzy matching tools (e.g., RapidFuzz), and entity linking (`entity_linking.py`).
    - **Containerization:** Docker or Singularity containers to ensure consistent processing environments.

B. Security Best Practices

- **Secret Management:**
    - Store API keys and tokens securely using environment variables or secret management tools (e.g., HashiCorp Vault).
- **Least Privilege:**
    - Grant the minimal necessary permissions to API endpoints and database operations.
- **Regular Audits:**
    - Conduct periodic security audits to identify and mitigate vulnerabilities.

C. References

- ngrok Documentation
- Flask RESTful APIs
- MongoDB Transactions
- Docker Best Practices
- spaCy NER Documentation
- RapidFuzz Documentation
- OpenAI API Documentation

Implementation Scripts

Below are the necessary scripts to implement the automated offloading of NER and fuzzy matching processing from the local server to the HPC cluster. The system is designed to be modular and extensible, allowing for the addition of new processing tasks in the future.

1. Local Server Scripts

1.1 Docker Compose Configuration (`docker-compose.yml`)

Sets up the MongoDB and Flask application containers with health checks, logging configurations, and network settings.

```
yaml
Copy code
```

```
version: '3.8'

services:
  mongodb:
    image: mongo:6.0
    container_name: mongodb
    environment:
      MONGO_INITDB_ROOT_USERNAME: admin
      MONGO_INITDB_ROOT_PASSWORD: secret
    volumes:
      - mongodb_data:/data/db
      - ./mongo-init/:/docker-entrypoint-initdb.d/
    ports:
      - "27017:27017"
    healthcheck:
      test: [
        "CMD",
        "mongosh",
        "--username",
        "admin",
        "--password",
        "secret",
        "--authenticationDatabase",
        "admin",
        "--eval",
        "db.adminCommand('ping')"
      ]
      interval: 10s
      timeout: 5s
      retries: 5
    networks:
      - app_network
    logging:
      driver: "json-file"
      options:
        max-size: "10m"
        max-file: "3"

  app:
    build:
      context: ./app
      dockerfile: Dockerfile
    container_name: flask_app
    ports:
      - "5000:5000"
    environment:
      MONGO_URI:
"mongodb://${MONGO_INITDB_ROOT_USERNAME}:${MONGO_INITDB_ROOT_PASSWORD}@mongod
b:27017/admin"
      FLASK_APP: app.py
      FLASK_ENV: ${FLASK_ENV}
      FLASK_DEBUG: ${FLASK_DEBUG}
      SECRET_KEY: ${SECRET_KEY}
    volumes:
      - ./app:/app
      - ./archives:/app/archives
    depends_on:
```

```
      mongodb:
        condition: service_healthy
    networks:
      - app_network
    logging:
      driver: "json-file"
      options:
        max-size: "10m"
        max-file: "3"

volumes:
  mongodb_data:

networks:
  app_network:
    driver: bridge

# Notes:
# - Ensure you have a .env file in your project root with the necessary
variables.
# - Add .env to your .gitignore to prevent committing sensitive information.
```

1.2 Multi-Stage Dockerfile for Flask App (`app/Dockerfile`)

Optimizes the Docker image by separating the build and runtime environments and pre-downloading spaCy models.

```
dockerfile
Copy code
# Stage 1: Build
FROM python:3.10-slim AS builder

# Set environment variables to prevent .pyc files and ensure output is
flushed
ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1

# Set the working directory inside the container
WORKDIR /app

# Install system dependencies required for building Python packages
RUN apt-get update && apt-get install -y --no-install-recommends \
    build-essential \
    curl \
    && rm -rf /var/lib/apt/lists/*

# Copy the requirements file and install Python dependencies
COPY requirements.txt /app/
RUN pip install --upgrade pip && pip install --user -r requirements.txt

# Create the directory for spaCy models to ensure it exists
RUN mkdir -p /root/.local/share/spacy/models

# Download spaCy models only if they are not already downloaded
RUN if [ ! -d "/root/.local/share/spacy/models/en_core_web_lg" ]; then \
```

```
        python -m spacy download en_core_web_lg; \
    fi && \
    if [ ! -d "/root/.local/share/spacy/models/en_core_web_trf" ]; then \
        python -m spacy download en_core_web_trf; \
    fi

# Copy project files into the container
COPY . .

# Stage 2: Runtime
FROM python:3.10-slim

# Set environment variables for the runtime stage
ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1

# Set the working directory inside the runtime container
WORKDIR /app

# Install only necessary system dependencies for the runtime
RUN apt-get update && apt-get install -y --no-install-recommends \
    && rm -rf /var/lib/apt/lists/*

# Copy installed Python packages and spaCy models from the builder stage
COPY --from=builder /root/.local /root/.local
COPY --from=builder /root/.local/share/spacy/models
/root/.local/share/spacy/models

# Set PATH to include user-installed binaries
ENV PATH=/root/.local/bin:$PATH

# Copy project files from the builder stage
COPY --from=builder /app /app

# Copy and set permissions for the entrypoint script
COPY entrypoint.sh /app/entrypoint.sh
RUN chmod +x /app/entrypoint.sh

# Set the entrypoint to the entrypoint script for container initialization
ENTRYPOINT ["/app/entrypoint.sh"]

# Command to run the Flask app
CMD ["python", "app.py"]
```

1.3 Entrypoint Script (`entrypoint.sh`)

Handles service initialization, including waiting for MongoDB to be ready before starting the Flask application.

```
bash
Copy code
#!/bin/bash

# Exit immediately if a command exits with a non-zero status
set -e
```

```
# Configuration for backoff
MAX_RETRIES=10
SLEEP_TIME=10
MONGO_HOST="mongodb"
MONGO_PORT=27017
MONGO_URI="mongodb://admin:secret@${MONGO_HOST}:${MONGO_PORT}/admin"

echo "Waiting for MongoDB to be ready..."

# Backoff loop to wait for MongoDB
for ((i=1;i<=MAX_RETRIES;i++)); do
    echo "Attempt $i/$MAX_RETRIES: Checking MongoDB connection..."
    python -c "import pymongo; client = pymongo.MongoClient('${MONGO_URI}',
serverSelectionTimeoutMS=5000); client.admin.command('ping')" && break
    echo "MongoDB is not ready yet. Waiting ${SLEEP_TIME} seconds..."
    sleep ${SLEEP_TIME}
done

# Verify if MongoDB is up after retries
python -c "import pymongo; client = pymongo.MongoClient('${MONGO_URI}',
serverSelectionTimeoutMS=5000); client.admin.command('ping')" || { echo
"MongoDB did not become ready in time after ${MAX_RETRIES} attempts.
Exiting."; exit 1; }

echo "MongoDB is up and running."

# Optional: Run database setup and processing scripts
# Uncomment the following lines if needed
# echo "Running database_setup.py..."
# python database_setup.py
# echo "database_setup.py completed."

# echo "Running data_processing.py..."
# python data_processing.py
# echo "data_processing.py completed."

# echo "Running generate_unique_terms.py..."
# python generate_unique_terms.py
# echo "generate_unique_terms.py completed."

# echo "Starting Flask app..."
exec "$@"

# Testing Version (Optional)
# Uncomment for profiling purposes
# echo "***Running testing version***"
# python -m cProfile -o show_env.prof show_env.py
# python -m cProfile -o database_setup.prof database_setup.py
# python -m cProfile -o data_processing.prof data_processing.py
# python -m cProfile -o generate_unique_terms.prof generate_unique_terms.py
# echo "Setup scripts completed. Starting Flask app..."
```

1.4 Flask Routes (`app/routes.py`)

Implements the Flask routes with enhanced security, logging, and error handling.

```
python
Copy code
# routes.py

from flask import request, jsonify, render_template, redirect, url_for,
flash, session, abort, Response, send_file
from functools import wraps
from app import app, cache
from database_setup import (
    get_client,
    get_db,
    get_collections,
    find_document_by_id,
    update_document,
    delete_document,
    get_field_structure
)
from bson import ObjectId
from werkzeug.security import generate_password_hash, check_password_hash
import math
import json
import re
import logging
import time
from datetime import datetime, timedelta
import random
import csv
from io import StringIO
from logging.handlers import RotatingFileHandler
import os
import uuid
import pymongo

# Create a logger instance
logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)

# Define a log format
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s -
%(message)s')

# Create a console handler (optional)
console_handler = logging.StreamHandler()
console_handler.setLevel(logging.DEBUG)
console_handler.setFormatter(formatter)
logger.addHandler(console_handler)

# Create a file handler to log to routes.log
log_file_path = os.path.join(os.path.dirname(__file__), 'routes.log')
file_handler = logging.FileHandler(log_file_path)
file_handler.setLevel(logging.DEBUG)  # Set the level for the file handler
file_handler.setFormatter(formatter)
logger.addHandler(file_handler)

# Initialize database connection and collections
client = get_client()
db = get_db(client)
```

```
documents, unique_terms_collection, field_structure_collection =
get_collections(db)

# Hashed password (generate this using
generate_password_hash('your_actual_password'))
ADMIN_PASSWORD_HASH =
'pbkdf2:sha256:260000$uxZ1Fkjt9WQCHwuN$ca37dfb41ebc26b19daf24885ebcd09f607cab
85f92dcab13625627fd9ee902a'

# Login attempt tracking
MAX_ATTEMPTS = 5
LOCKOUT_TIME = 15 * 60  # 15 minutes in seconds
login_attempts = {}

def is_locked_out(ip):
    if ip in login_attempts:
        attempts, last_attempt_time = login_attempts[ip]
        if attempts >= MAX_ATTEMPTS:
            if datetime.now() - last_attempt_time <
timedelta(seconds=LOCKOUT_TIME):
                return True
            else:
                login_attempts[ip] = (0, datetime.now())
    return False

def update_login_attempts(ip, success):
    if ip in login_attempts:
        attempts, _ = login_attempts[ip]
        if success:
            login_attempts[ip] = (0, datetime.now())
        else:
            login_attempts[ip] = (attempts + 1, datetime.now())
    else:
        login_attempts[ip] = (0, datetime.now()) if success else (1,
datetime.now())

# Login required decorator
def login_required(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        if 'logged_in' not in session:
            return redirect(url_for('login', next=request.url))
        return f(*args, **kwargs)
    return decorated_function

@app.route('/')
# @login_required
def index():
    app.logger.info('Handling request to index')
    num_search_fields = 3  # Number of search fields to display
    field_structure = get_field_structure(db)  # Pass 'db' here
    return render_template('index.html', num_search_fields=num_search_fields,
field_structure=field_structure)

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
```

```
        ip = request.remote_addr

        if is_locked_out(ip):
            flash('Too many failed attempts. Please try again later.')
            return render_template('login.html')

        # Verify CAPTCHA
        user_captcha = request.form.get('captcha')
        correct_captcha = request.form.get('captcha_answer')
        if user_captcha != correct_captcha:
            flash('Incorrect CAPTCHA')
            return redirect(url_for('login'))

        if check_password_hash(ADMIN_PASSWORD_HASH,
request.form['password']):
            session['logged_in'] = True
            update_login_attempts(ip, success=True)
            flash('You were successfully logged in')
            next_page = request.args.get('next')
            return redirect(next_page or url_for('index'))
        else:
            update_login_attempts(ip, success=False)
            time.sleep(2)  # Add a delay after failed attempt
            flash('Invalid password')

    # Generate CAPTCHA for GET requests
    captcha_num1 = random.randint(1, 10)
    captcha_num2 = random.randint(1, 10)
    captcha_answer = str(captcha_num1 + captcha_num2)

    return render_template('login.html', captcha_num1=captcha_num1,
captcha_num2=captcha_num2, captcha_answer=captcha_answer)

@app.route('/logout')
def logout():
    session.pop('logged_in', None)
    flash('You were logged out')
    return redirect(url_for('index'))

@app.route('/search', methods=['POST'])
# @login_required
def search():
    try:
        data = request.get_json()
        logger.debug(f"Received search request: {data}")

        page = int(data.get('page', 1))
        per_page = int(data.get('per_page', 50))

        query = build_query(data)
        logger.debug(f"Constructed MongoDB query: {query}")

        total_count = documents.count_documents(query)
        search_results = list(documents.find(query).skip((page - 1) *
per_page).limit(per_page))

        for doc in search_results:
```

```
            doc['_id'] = str(doc['_id'])

        total_pages = math.ceil(total_count / per_page) if per_page else 1

        # Generate unique search ID
        search_id = str(uuid.uuid4())
        # Store the ordered list of document IDs
        ordered_ids = [doc['_id'] for doc in search_results]
        cache.set(f'search_{search_id}', ordered_ids, timeout=3600)  #
Expires in 1 hour

        logger.debug(f"Search ID: {search_id}, Found {total_count}
documents.")

        return jsonify({
            "search_id": search_id,
            "documents": search_results,
            "total_count": total_count,
            "current_page": page,
            "total_pages": total_pages,
            "per_page": per_page
        })
    except Exception as e:
        logger.error(f"An error occurred during search: {str(e)}",
exc_info=True)
        return jsonify({"error": "An internal error occurred"}), 500

def build_query(data):
    query = {}
    criteria_list = []

    logger.debug(f"Building query from search data: {data}")

    for i in range(1, 4):
        field = data.get(f'field{i}')
        search_term = data.get(f'searchTerm{i}')
        operator = data.get(f'operator{i}')

        if field and search_term:
            condition = {}
            if operator == 'NOT':
                condition[field] = {'$not': {'$regex':
re.escape(search_term), '$options': 'i'}}
            else:
                condition[field] = {'$regex': re.escape(search_term),
'$options': 'i'}

            criteria_list.append((operator, condition))
            logger.debug(f"Processed field {field} with search term
'{search_term}' and operator '{operator}'")

    if criteria_list:
        and_conditions = []
        or_conditions = []

        for operator, condition in criteria_list:
            if operator == 'AND' or operator == 'NOT':
```

```
                and_conditions.append(condition)
            elif operator == 'OR':
                or_conditions.append(condition)

        if and_conditions:
            query['$and'] = and_conditions

        if or_conditions:
            if '$or' not in query:
                query['$or'] = or_conditions
            else:
                query['$or'].extend(or_conditions)

    logger.debug(f"Final query: {query}")
    return query

@app.route('/document/<string:doc_id>')
# @login_required
def document_detail(doc_id):
    # Hard-coded SHOW_EMPTY variable
    SHOW_EMPTY = False  # Set to True to show empty fields, False to hide
them

    # Function to clean the document data
    def clean_data(data):
        empty_values = [None, '', 'N/A', 'null', [], {}, 'None']
        if isinstance(data, dict):
            return {
                k: clean_data(v)
                for k, v in data.items()
                if v not in empty_values and clean_data(v) not in
empty_values
            }
        elif isinstance(data, list):
            return [
                clean_data(item)
                for item in data
                if item not in empty_values and clean_data(item) not in
empty_values
            ]
        else:
            return data

    search_id = request.args.get('search_id')
    if not search_id:
        flash('Missing search context.')
        return redirect(url_for('index'))

    try:
        # Fetch the document by ID
        document = find_document_by_id(db, doc_id)
        if not document:
            abort(404)

        document['_id'] = str(document['_id'])

        # Log the document information for debugging
```

```
        logger.debug(f"Retrieved document for ID {doc_id}: {document}")

        # Decide whether to clean the document based on SHOW_EMPTY
        if SHOW_EMPTY:
            document = document
        else:
            # Clean the document to remove empty fields
            document = clean_data(document)

        # Retrieve the ordered list from cache
        ordered_ids = cache.get(f'search_{search_id}')
        if not ordered_ids:
            flash('Search context expired. Please perform the search again.')
            return redirect(url_for('index'))

        try:
            current_index = ordered_ids.index(doc_id)
        except ValueError:
            flash('Document not found in the current search results.')
            return redirect(url_for('index'))

        # Determine previous and next IDs based on the search order
        prev_id = ordered_ids[current_index - 1] if current_index > 0 else
None
        next_id = ordered_ids[current_index + 1] if current_index <
len(ordered_ids) - 1 else None

        # Get the relative path from the document
        relative_path = document.get('relative_path')  # This should contain
the relative path to the JSON file

        if relative_path:
            # Construct the image path by removing the '.json' extension
            image_path = relative_path.replace('.json', '')  # e.g.,
'rolls/rolls/tray_1_roll_5_page3303_img1.png'
            logger.debug(f"Document ID: {doc_id}, Image path: {image_path}")

            # Check if the image file exists
            absolute_image_path = os.path.join('/app/archives', image_path)
            image_exists = os.path.exists(absolute_image_path)

            if not image_exists:
                logger.warning(f"Image not found at: {absolute_image_path}")
        else:
            # Log an error if relative_path is None or not found
            logger.error(f"Error: No relative_path found for document ID:
{doc_id}. Document content: {document}")
            image_exists = False
            image_path = None

        # Render the template with all required variables
        return render_template(
            'document-detail.html',
            document=document,
            prev_id=prev_id,
            next_id=next_id,
            search_id=search_id,
```

```
            image_path=image_path,  # Pass the constructed image path
            image_exists=image_exists  # Pass the flag indicating if the
image exists
        )
    except Exception as e:
        logger.error(f"Error in document_detail: {str(e)}", exc_info=True)
        abort(500)

@app.route('/images/<path:filename>')
def serve_image(filename):
    image_path = os.path.join('/app/archives', filename)
    logger.debug(f"Serving image from: {image_path}")
    if os.path.exists(image_path):
        return send_file(image_path)
    else:
        logger.warning(f"Image not found at: {image_path}")
        abort(404)

def get_top_unique_terms(db, field, term_type, query='', limit=1000, skip=0):
    """
    Retrieve top unique terms based on the field, term type, and optional
search query.

    :param db: Database instance
    :param field: The field to filter terms by (e.g., 'title', 'description')
    :param term_type: The type of term ('word' or 'phrase')
    :param query: Optional search query string to filter terms
    :param limit: Number of top terms to retrieve
    :param skip: Number of records to skip for pagination
    :return: List of dictionaries with term and count
    """
    unique_terms_collection = db['unique_terms']

    try:
        # Base MongoDB query
        mongo_query = {"field": field, "type": term_type}

        # If a search query is provided, add a regex filter for the 'term'
field
        if query:
            # Escape special regex characters to prevent injection attacks
            escaped_query = re.escape(query)
            # Case-insensitive search for terms containing the query
substring
            mongo_query['term'] = {"$regex": f".*{escaped_query}.*",
"$options": "i"}

        start_time = time.time()

        # Execute the query with sorting, skipping, and limiting for
pagination
        cursor = unique_terms_collection.find(
            mongo_query,
            {"_id": 0, "term": 1, "frequency": 1}
        ).sort("frequency", pymongo.DESCENDING).skip(skip).limit(limit)

        terms_list = []
```

```
        for doc in cursor:
            key = 'word' if term_type == 'word' else 'phrase'
            terms_list.append({key: doc['term'], 'count': doc['frequency']})

        duration = time.time() - start_time
        logger.info(f"Retrieved top {len(terms_list)} {term_type}s in
{duration:.4f} seconds for field '{field}' with query '{query}'.")

        return terms_list
    except Exception as e:
        logger.error(f"Error retrieving unique terms: {e}")
        return []

def get_unique_terms_count(db, field, term_type, query=''):
    """
    Get the count of unique terms based on the field, term type, and optional
search query.

    :param db: Database instance
    :param field: The field to filter terms by
    :param term_type: The type of term ('word' or 'phrase')
    :param query: Optional search query string to filter terms
    :return: Integer count of unique terms
    """
    unique_terms_collection = db['unique_terms']

    try:
        # Base MongoDB query
        mongo_query = {"field": field, "type": term_type}

        # If a search query is provided, add a regex filter for the 'term'
field
        if query:
            # Escape special regex characters to prevent injection attacks
            escaped_query = re.escape(query)
            # Case-insensitive search for terms containing the query
substring
            mongo_query['term'] = {"$regex": f".*{escaped_query}.*",
"$options": "i"}

        # Count the number of unique terms matching the query
        count = unique_terms_collection.count_documents(mongo_query)
        logger.info(f"Counted {count} unique {term_type}s for field '{field}'
with query '{query}'.")
        return count
    except Exception as e:
        logger.error(f"Error counting unique terms: {e}")
        return 0

@app.route('/search-terms', methods=['GET'])
def search_terms():
    client = get_client()  # Initialize your MongoDB client
    db = get_db(client)    # Get the database instance

    if request.headers.get('X-Requested-With') == 'XMLHttpRequest':
        # Handle AJAX request
        field = request.args.get('field')
```

```
        if not field:
            return jsonify({"error": "No field specified"}), 400

        # Extract the search query
        query = request.args.get('query', '').strip().lower()  # Normalize
the query

        logger.debug(f"AJAX request for field: {field}, query: {query}")  #
Using logger here

        # Define term types
        term_types = ['word', 'phrase']
        data = {}
        total_records = 0

        for term_type in term_types:
            page = int(request.args.get('page', 1))
            per_page = int(request.args.get('per_page', 100))
            skip = (page - 1) * per_page

            # Fetch filtered terms based on the query
            terms = get_top_unique_terms(db, field, term_type, query=query,
limit=per_page, skip=skip)
            data[term_type + 's'] = terms

            # Fetch the count of unique terms based on the query
            count = get_unique_terms_count(db, field, term_type, query=query)
            data['unique_' + term_type + 's'] = count
            total_records += count

        data['total_records'] = total_records

        return jsonify(data)
    else:
        # Render the HTML template
        field_structure = get_field_structure(db)
        unique_fields = []  # Define if necessary
        return render_template('search-terms.html',
field_structure=field_structure, unique_fields=unique_fields)

@app.route('/database-info')
# @login_required
def database_info():
    field_struct = get_field_structure(db)  # Pass 'db' here
    collection_info = []

    def count_documents_with_field(field_path):
        count = documents.count_documents({field_path: {'$exists': True}})
        return count

    def traverse_structure(structure, current_path=''):
        for field, value in structure.items():
            path = f"{current_path}.{field}" if current_path else field
            if isinstance(value, dict):
                traverse_structure(value, current_path=path)
            else:
                count = count_documents_with_field(path)
```

```
                collection_info.append({
                    'name': path,
                    'count': count
                })

    traverse_structure(field_struct)

    return render_template('database-info.html',
collection_info=collection_info)

@app.route('/settings', methods=['GET', 'POST'])
# @login_required
def settings():
    config_path = os.path.join(os.path.dirname(__file__), 'config.json')

    if request.method == 'POST':
        new_config = request.form.to_dict()

        for key in ['fonts', 'sizes', 'colors', 'spacing']:
            if key in new_config:
                try:
                    new_config[key] = json.loads(new_config[key])
                except json.JSONDecodeError:
                    flash(f"Invalid JSON format for {key}.", 'danger')
                    return redirect(url_for('settings'))

        try:
            with open(config_path, 'w') as config_file:
                json.dump(new_config, config_file, indent=4)
            app.config['UI_CONFIG'] = new_config
            flash('Settings updated successfully', 'success')
        except Exception as e:
            logger.error(f"Error updating settings: {str(e)}")
            flash('Failed to update settings.', 'danger')
        return redirect(url_for('settings'))

    try:
        if os.path.exists(config_path):
            with open(config_path) as config_file:
                config = json.load(config_file)
        else:
            config = {}
    except json.JSONDecodeError:
        config = {}
        flash('Configuration file is corrupted. Using default settings.',
'warning')

    return render_template('settings.html', config=config)

# Consider streaming if it ends up being thousands of documents
@app.route('/export_selected_csv', methods=['POST'])
# @login_required
def export_selected_csv():
    try:
        data = request.get_json()
        document_ids = data.get('document_ids', [])
        if not document_ids:
```

```
            return jsonify({"error": "No document IDs provided"}), 400

        # Convert string IDs to ObjectIds, handle invalid IDs
        valid_ids = []
        for doc_id in document_ids:
            try:
                valid_ids.append(ObjectId(doc_id))
            except Exception as e:
                logger.warning(f"Invalid document ID: {doc_id}")

        if not valid_ids:
            return jsonify({"error": "No valid document IDs provided"}), 400

        # Check if any documents exist with the provided IDs
        count = documents.count_documents({"_id": {"$in": valid_ids}})
        if count == 0:
            return jsonify({"error": "No documents found for the provided
IDs."}), 404

        # Retrieve the documents
        documents_cursor = documents.find({"_id": {"$in": valid_ids}})

        # Create CSV
        output = StringIO()
        writer = csv.writer(output)
        writer.writerow(['filename', 'OCR', 'original_json'])  # Header row

        for doc in documents_cursor:
            filename = doc.get('filename', 'N/A')
            ocr = doc.get('summary', 'N/A')  # Adjust field as necessary
            original_json = json.dumps(doc, default=str)  # Convert ObjectId
to string if necessary
            writer.writerow([filename, ocr, original_json])

        # Prepare CSV for download
        output.seek(0)
        return Response(
            output.getvalue(),
            mimetype='text/csv',
            headers={'Content-Disposition': 'attachment;
filename=selected_documents.csv'}
        )

    except Exception as e:
        logger.error(f"Error exporting selected CSV: {str(e)}",
exc_info=True)
        return jsonify({"error": "An internal error occurred"}), 500

@app.errorhandler(404)
def not_found_error(error):
    return render_template('error.html', message='Page not found'), 404

@app.errorhandler(500)
def internal_error(error):
    return render_template('error.html', message='An unexpected error has
occurred'), 500
```

1.5 Flask Application Initialization (`app/app.py`)

Initializes the Flask application, cache, and database connections.

```
python
Copy code
# app/app.py

from flask import Flask
from flask_caching import Cache
from database_setup import get_client, get_db, get_collections

app = Flask(__name__)
app.config.from_envvar('APP_CONFIG_FILE')  # Ensure the environment variable
points to your config file

# Initialize Cache (e.g., SimpleCache or RedisCache for production)
cache = Cache(app, config={'CACHE_TYPE': 'simple'})

# Initialize database connection
client = get_client()
db = get_db(client)
documents, unique_terms_collection, field_structure_collection =
get_collections(db)

# Import routes after initializing app and db
from . import routes
```

1.6 Database Setup Script (`app/database_setup.py`)

Sets up MongoDB collections, indexes, and provides utility functions for database interactions.

```
python
Copy code
# app/database_setup.py

import os
import logging
from pymongo import MongoClient, ASCENDING, DESCENDING
from pymongo.errors import ConnectionFailure

# Configure logging
logger = logging.getLogger('DatabaseSetup')
logger.setLevel(logging.DEBUG)
if not logger.handlers:
    console_handler = logging.StreamHandler()
    console_handler.setLevel(logging.DEBUG)
    formatter = logging.Formatter('%(asctime)s - %(levelname)s -
%(message)s')
    console_handler.setFormatter(formatter)
    logger.addHandler(console_handler)

def get_client():
    """Initialize and return a new MongoDB client."""
```

```
    try:
        mongo_uri = os.environ.get('MONGO_URI')
        if not mongo_uri:
            raise ValueError("MONGO_URI environment variable not set")
        client = MongoClient(mongo_uri, serverSelectionTimeoutMS=5000)
        # Test connection
        client.admin.command('ping')
        logger.info("Successfully connected to MongoDB.")
        return client
    except Exception as e:
        logger.error(f"Failed to connect to MongoDB: {e}")
        raise e

def get_db(client):
    """Return the database instance."""
    return client['railroad_documents']

def get_collections(db):
    """Retrieve and return references to the required collections."""
    try:
        documents_collection = db['documents']
        unique_terms_collection = db['unique_terms']
        field_structure_collection = db['field_structure']
        return documents_collection, unique_terms_collection,
field_structure_collection
    except Exception as e:
        logger.error(f"Error getting collections: {e}")
        raise e

def initialize_database(client):
    """Initialize database collections, indexes, and any necessary setup."""
    db = get_db(client)
    documents, unique_terms_collection, field_structure_collection =
get_collections(db)

    # Create indexes for the documents collection
    documents.create_index([('filename', ASCENDING)], unique=True)
    documents.create_index([('ocr_text', 'text'), ('summary', 'text')])

    # Create indexes for unique_terms collection
    unique_terms_collection.create_index([('term', ASCENDING)], unique=True)
    unique_terms_collection.create_index([('field', ASCENDING), ('type',
ASCENDING)])

    # Create indexes for field_structure collection
    field_structure_collection.create_index([('field', ASCENDING)],
unique=True)

    logger.info("Database initialized with necessary collections and
indexes.")

def find_document_by_id(db, doc_id):
    """Find a document by its ID."""
    try:
        documents, _, _ = get_collections(db)
        document = documents.find_one({"_id": ObjectId(doc_id)})
        return document
```

```
    except Exception as e:
        logger.error(f"Error finding document by ID {doc_id}: {e}")
        return None

def update_document(db, doc_id, update_fields):
    """Update a document with given fields."""
    try:
        documents, _, _ = get_collections(db)
        result = documents.update_one({"_id": ObjectId(doc_id)}, {"$set":
update_fields})
        return result.modified_count
    except Exception as e:
        logger.error(f"Error updating document {doc_id}: {e}")
        return 0

def delete_document(db, doc_id):
    """Delete a document by its ID."""
    try:
        documents, _, _ = get_collections(db)
        result = documents.delete_one({"_id": ObjectId(doc_id)})
        return result.deleted_count
    except Exception as e:
        logger.error(f"Error deleting document {doc_id}: {e}")
        return 0

def get_field_structure(db):
    """Retrieve the field structure of the documents."""
    try:
        _, _, field_structure_collection = get_collections(db)
        structure = field_structure_collection.find_one({})
        return structure.get('structure', {}) if structure else {}
    except Exception as e:
        logger.error(f"Error retrieving field structure: {e}")
        return {}
```

1.7 Data Processing Script (`app/data_processing.py`)

Processes incoming JSON and TXT files, inserting them into MongoDB and updating field structures.

```
python
Copy code
# app/data_processing.py

import os
import json
import logging
from database_setup import get_client, get_db, get_collections
from bson.objectid import ObjectId
from json_validator import validate_json  # Assume you have a JSON validator
script

# Configure logging
logger = logging.getLogger('DataProcessing')
logger.setLevel(logging.DEBUG)
```

```
if not logger.handlers:
    console_handler = logging.StreamHandler()
    console_handler.setLevel(logging.DEBUG)
    formatter = logging.Formatter('%(asctime)s - %(levelname)s -
%(message)s')
    console_handler.setFormatter(formatter)
    logger.addHandler(console_handler)

def process_files(directory):
    """Process JSON and TXT files in the given directory."""
    client = get_client()
    db = get_db(client)
    documents, unique_terms_collection, field_structure_collection =
get_collections(db)

    for filename in os.listdir(directory):
        filepath = os.path.join(directory, filename)
        if not os.path.isfile(filepath):
            continue

        if filename.endswith('.json'):
            with open(filepath, 'r', encoding='utf-8') as file:
                try:
                    data = json.load(file)
                    if not validate_json(data):
                        logger.warning(f"Validation failed for {filename}.
Skipping.")
                        continue

                    # Insert document into MongoDB
                    result = documents.insert_one(data)
                    logger.info(f"Inserted document {result.inserted_id} from
{filename}.")

                except json.JSONDecodeError:
                    logger.error(f"Invalid JSON format in {filename}.
Skipping.")
                except Exception as e:
                    logger.error(f"Error processing {filename}: {e}")

        elif filename.endswith('.txt'):
            # Handle TXT files if necessary
            logger.info(f"Processing TXT file: {filename}")
            # Implement TXT processing logic here

def main():
    """Main function to process files."""
    archives_dir = os.path.join(os.getcwd(), 'archives')
    if not os.path.exists(archives_dir):
        logger.error(f"Archives directory does not exist: {archives_dir}")
        return

    process_files(archives_dir)
    logger.info("Data processing completed.")

if __name__ == "__main__":
    main()
```

1.8 NER Processing Script (`app/ner_processing.py`)

Handles NER processing on documents, extracting entities and preparing them for linking.

```
python
Copy code
# app/ner_processing.py

import os
import logging
from collections import defaultdict
from database_setup import get_client, get_db, get_collections
from ner_worker import process_documents_batch, initialize_spacy
from entity_linking import link_entities

# Configure logging
logger = logging.getLogger('NERProcessing')
logger.setLevel(logging.DEBUG)
if not logger.handlers:
    console_handler = logging.StreamHandler()
    console_handler.setLevel(logging.DEBUG)
    formatter = logging.Formatter('%(asctime)s - %(levelname)s -
%(message)s')
    console_handler.setFormatter(formatter)
    logger.addHandler(console_handler)

def main():
    """Main function to perform NER and fuzzy matching."""
    client = get_client()
    db = get_db(client)
    documents, linked_entities_collection, _ = get_collections(db)

    # Define processing parameters
    FIELDS_TO_PROCESS = ["ocr_text", "summary"]
    LINK_WIKIDATA = False
    FUZZY_MATCH_THRESHOLD = 90
    BATCH_SIZE = 1000
    ENABLE_MULTIPROCESSING = True

    # Perform entity extraction and linking
    extract_and_link_entities(
        documents_collection=documents,
        linked_entities_collection=linked_entities_collection,
        fields_to_process=FIELDS_TO_PROCESS,
        link_wikidata=LINK_WIKIDATA,
        fuzzy_threshold=FUZZY_MATCH_THRESHOLD,
        batch_size=BATCH_SIZE,
        use_multiprocessing=ENABLE_MULTIPROCESSING
    )

if __name__ == "__main__":
    main()
```

1.9 Entity Linking Script (`entity_linking.py`)

This script handles the linking of extracted entities to external knowledge bases (e.g., Wikidata) and updates the MongoDB collection with the linked entities.

```
python
Copy code
# entity_linking.py

import os
import logging
import requests
from pymongo import UpdateOne, MongoClient
from dotenv import load_dotenv
from collections import defaultdict
from rapidfuzz import process, fuzz

# Load environment variables from .env file
load_dotenv()

# =======================
# Logging Configuration
# =======================
logger = logging.getLogger('EntityLinkingLogger')
logger.setLevel(logging.DEBUG)

if not logger.handlers:
    console_handler = logging.StreamHandler()
    console_handler.setLevel(logging.DEBUG)

    file_handler = logging.FileHandler('entity_linking.log', mode='a')
    file_handler.setLevel(logging.DEBUG)

    formatter = logging.Formatter('%(asctime)s - %(levelname)s -
%(message)s')
    console_handler.setFormatter(formatter)
    file_handler.setFormatter(formatter)

    logger.addHandler(console_handler)
    logger.addHandler(file_handler)

# =======================
# Database Functions
# =======================

def get_client():
    """Initialize and return a new MongoDB client."""
    try:
        mongo_uri = os.environ.get('MONGO_URI')
        if not mongo_uri:
            raise ValueError("MONGO_URI environment variable not set")
        client = MongoClient(mongo_uri, serverSelectionTimeoutMS=5000)
        # Test connection
        client.admin.command('ping')
        logger.info("Successfully connected to MongoDB.")
```

```
        return client
    except Exception as e:
        logger.error(f"Failed to connect to MongoDB: {e}")
        raise e

def get_db(client):
    """Return the database instance."""
    return client['railroad_documents']

def get_collections(db):
    """Retrieve and return references to the required collections."""
    try:
        documents_collection = db['documents']
        linked_entities_collection = db['linked_entities']
        return documents_collection, linked_entities_collection
    except Exception as e:
        logger.error(f"Error getting collections: {e}")
        raise e

# ======================
# Utility Functions
# ======================

def fuzzy_match(term, reference_terms, threshold=90):
    """
    Perform fuzzy matching to find the best match for a term in
reference_terms.
    Returns the matched term if similarity exceeds the threshold, else None.
    """
    try:
        result = process.extractOne(term, reference_terms,
scorer=fuzz.token_sort_ratio)
        if result is None:
            logger.debug(f"No fuzzy match found for term '{term}'.")
            return None

        match, score = result
        logger.debug(f"Fuzzy match for term '{term}': '{match}' with score
{score}.")

        if score >= threshold:
            return match
        return None
    except Exception as e:
        logger.error(f"Exception during fuzzy matching for term '{term}':
{e}")
        return None

def fetch_wikidata_entity(term):
    """
    Fetch Wikidata entity ID for a given term using the Wikidata API.
    Returns the entity ID if found, else None.
    """
    wikidata_cache = {}
    if term in wikidata_cache:
        return wikidata_cache[term]
    try:
```

```
        url = "https://www.wikidata.org/w/api.php"
        params = {
            'action': 'wbsearchentities',
            'search': term,
            'language': 'en',
            'format': 'json'
        }
        response = requests.get(url, params=params, timeout=5)
        data = response.json()
        if 'search' in data and len(data['search']) > 0:
            # Return the first matching entity ID
            wikidata_id = data['search'][0]['id']
            wikidata_cache[term] = wikidata_id
            logger.debug(f"Fetched Wikidata ID '{wikidata_id}' for term
'{term}'.")
            return wikidata_id
        else:
            wikidata_cache[term] = None
            logger.debug(f"No Wikidata ID found for term '{term}'.")
            return None
    except Exception as e:
        logger.error(f"Error fetching Wikidata entity for term '{term}':
{e}")
        return None

# ======================
# Entity Linking Function
# ======================

def link_entities(aggregated_entities, linked_entities_collection,
existing_linked_terms, link_wikidata, fuzzy_threshold, batch_size=1000):
    """
    Link aggregated entities and update the linked_entities collection.
    """
    linked_count = 0
    processed_entities = 0
    total_entities = len(aggregated_entities)
    logger.info(f"Total unique entities to process: {total_entities}")

    operations = []

    for (term_lower, ent_type), data in aggregated_entities.items():
        frequency = data['frequency']
        document_ids = list(data['document_ids'])
        wikidata_id = None

        if link_wikidata:
            wikidata_id = fetch_wikidata_entity(term_lower)

        # If Wikidata linking is disabled or Wikidata ID not found, perform
fuzzy matching
        if not wikidata_id:
            fuzzy_match_term = fuzzy_match(term_lower, existing_linked_terms,
threshold=fuzzy_threshold)
            if fuzzy_match_term:
                # Fetch the corresponding Wikidata ID for the matched term
```

```
                matched_entity = linked_entities_collection.find_one({"term":
fuzzy_match_term.lower()})
                if matched_entity:
                    wikidata_id = matched_entity.get('kb_id')
                    logger.debug(f"Fuzzy matched term '{term_lower}' to
'{fuzzy_match_term}' with Wikidata ID '{wikidata_id}'.")
            else:
                logger.debug(f"No match found for term '{term_lower}'.")

        if wikidata_id:
            logger.debug(f"Linked term '{term_lower}' to Wikidata ID
'{wikidata_id}'.")

        update = {
            "$inc": {"frequency": frequency},
            "$addToSet": {"document_ids": {"$each": document_ids}},
            "$set": {"type": ent_type, "kb_id": wikidata_id}
        }

        operations.append(
            UpdateOne(
                {"term": term_lower},
                update,
                upsert=True
            )
        )

        linked_count += 1
        processed_entities += 1

        # Log progress every 1000 entities
        if processed_entities % 1000 == 0:
            logger.info(f"Processed {processed_entities}/{total_entities}
entities.")

        # Execute operations in batches
        if len(operations) >= batch_size:
            logger.info(f"Writing batch of {len(operations)} entities to the
database.")
            try:
                result = linked_entities_collection.bulk_write(operations,
ordered=False)
                logger.info(f"Bulk upserted {result.upserted_count +
result.modified_count} linked entities.")
            except Exception as e:
                logger.error(f"Error bulk upserting linked entities: {e}")
                raise e
            operations = []

    # Execute remaining operations
    if operations:
        logger.info(f"Writing final batch of {len(operations)} entities to
the database.")
        try:
            result = linked_entities_collection.bulk_write(operations,
ordered=False)
```

```
            logger.info(f"Bulk upserted {result.upserted_count +
result.modified_count} linked entities.")
        except Exception as e:
            logger.error(f"Error bulk upserting linked entities: {e}")
            raise e

    logger.info(f"Successfully linked {linked_count} entities.")
```

1.10 API Integration Script (`api_integration.py`)

This script facilitates communication between the local server and the HPC cluster by defining additional API endpoints if necessary. It ensures that the system remains abstracted, allowing for future offloading of various computational tasks.

```
python
Copy code
# api_integration.py

from flask import Blueprint, request, jsonify
from functools import wraps
import os
import logging
import requests
import json
import gzip

# Create a Blueprint for API routes
api_bp = Blueprint('api', __name__)

# =======================
# Logging Configuration
# =======================
logger = logging.getLogger('APIIntegrationLogger')
logger.setLevel(logging.DEBUG)

if not logger.handlers:
    console_handler = logging.StreamHandler()
    console_handler.setLevel(logging.DEBUG)

    file_handler = logging.FileHandler('api_integration.log', mode='a')
    file_handler.setLevel(logging.DEBUG)

    formatter = logging.Formatter('%(asctime)s - %(levelname)s -
%(message)s')
    console_handler.setFormatter(formatter)
    file_handler.setFormatter(formatter)

    logger.addHandler(console_handler)
    logger.addHandler(file_handler)

# =======================
# Authentication Decorator
# =======================
```

```
API_KEY = os.getenv('API_KEY', 'default_api_key')

def require_api_key(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        key = request.headers.get('x-api-key')
        if key and key == API_KEY:
            return f(*args, **kwargs)
        else:
            logger.warning(f"Unauthorized access attempt from
{request.remote_addr}")
            return jsonify({"message": "Unauthorized"}), 401
    return decorated

# =======================
# API Endpoints
# =======================

@api_bp.route('/api/execute-task', methods=['POST'])
@require_api_key
def execute_task():
    """
    Endpoint to receive tasks from the HPC cluster for execution.
    Allows offloading of additional computational tasks.
    """
    try:
        if not request.is_json:
            logger.error("Received non-JSON data.")
            return jsonify({"message": "Invalid data format. JSON
expected."}), 400

        data = request.get_json()
        task_type = data.get('task_type')
        task_payload = data.get('payload')

        if not task_type or not task_payload:
            logger.error("Missing 'task_type' or 'payload' in the request.")
            return jsonify({"message": "Missing 'task_type' or 'payload'."}),
400

        # Handle different task types
        if task_type == 'ner_fuzzy':
            # Process NER and fuzzy matching
            from ner_processing import extract_and_link_entities  # Import
here to avoid circular imports
            from database_setup import get_client, get_db, get_collections

            client = get_client()
            db = get_db(client)
            documents_collection, linked_entities_collection =
get_collections(db)

            # Extract entities and perform linking
            extract_and_link_entities(
                documents_collection,
                linked_entities_collection,
```

```
                fields_to_process=task_payload.get('fields_to_process',
["ocr_text", "summary"]),
                link_wikidata=task_payload.get('link_wikidata', False),
                fuzzy_threshold=task_payload.get('fuzzy_threshold', 90),
                batch_size=task_payload.get('batch_size', 1000),
                use_multiprocessing=task_payload.get('use_multiprocessing',
False)
            )

            logger.info(f"Completed task_type: {task_type}")
            return jsonify({"status": "success", "message": f"Task
{task_type} completed."}), 200

        else:
            logger.error(f"Unsupported task_type: {task_type}")
            return jsonify({"message": f"Unsupported task_type:
{task_type}."}), 400

    except Exception as e:
        logger.error(f"Error executing task: {e}", exc_info=True)
        return jsonify({"message": "An error occurred while executing the
task."}), 500
```

Integration with `app.py`:

To incorporate the `api_integration.py` blueprint into your Flask application, update `app.py` as follows:

```
python
Copy code
# app.py

from flask import Flask
from api_integration import api_bp  # Import the blueprint

app = Flask(__name__)

# ... existing configurations and routes ...

# Register the API blueprint
app.register_blueprint(api_bp)

# ... rest of the app ...
```

1.11 Entry Point Script (`entrypoint.sh`)

This script initializes the application by ensuring that MongoDB is ready before starting the Flask app. It also includes provisions for running setup scripts and can be extended for additional initialization tasks.

```
bash
Copy code
```

```
#!/bin/bash

# Exit immediately if a command exits with a non-zero status
set -e

# Configuration for backoff
MAX_RETRIES=10
SLEEP_TIME=10
MONGO_HOST="mongodb"
MONGO_PORT=27017
MONGO_URI="mongodb://admin:secret@${MONGO_HOST}:${MONGO_PORT}/admin"

echo "Waiting for MongoDB to be ready..."

# Backoff loop to wait for MongoDB
for ((i=1;i<=MAX_RETRIES;i++)); do
    echo "Attempt $i/$MAX_RETRIES: Checking MongoDB connection..."
    python -c "import pymongo; client = pymongo.MongoClient('${MONGO_URI}',
serverSelectionTimeoutMS=5000); client.admin.command('ping')" && break
    echo "MongoDB is not ready yet. Waiting ${SLEEP_TIME} seconds..."
    sleep ${SLEEP_TIME}
done

# Verify if MongoDB is up after retries
python -c "import pymongo; client = pymongo.MongoClient('${MONGO_URI}',
serverSelectionTimeoutMS=5000); client.admin.command('ping')" || { echo
"MongoDB did not become ready in time after ${MAX_RETRIES} attempts.
Exiting."; exit 1; }

echo "MongoDB is up and running."

# Uncomment the following lines to run setup scripts
# echo "Running database_setup.py..."
# python database_setup.py
# echo "database_setup.py completed."

# echo "Running data_processing.py..."
# python data_processing.py
# echo "data_processing.py completed."

# echo "Running generate_unique_terms.py..."
# python generate_unique_terms.py
# echo "generate_unique_terms.py completed."

# Uncomment if you want to run NER processing at startup
# echo "Running ner_processing.py..."
# python ner_processing.py
# echo "ner_processing.py completed."

echo "Setup scripts completed. Starting Flask app..."

# Start the Flask app
exec "$@"
```

Explanation:

- **MongoDB Health Check:** The script attempts to connect to MongoDB using the provided URI. It retries the connection up to `MAX_RETRIES` times, waiting `SLEEP_TIME` seconds between attempts.
- **Setup Scripts:** Provision is made to run setup scripts (`database_setup.py`, `data_processing.py`, `generate_unique_terms.py`, `ner_processing.py`) before starting the Flask app. These can be uncommented as needed.
- **Starting the Flask App:** After ensuring MongoDB is ready and running setup scripts, the script starts the Flask application.

1.12 Environment Variables (`.env`)

Ensure that all necessary environment variables are defined in the `.env` file. This file should be located in the root directory and included in `.gitignore` to prevent sensitive information from being committed to version control.

```
env
Copy code
# MongoDB Credentials
MONGO_INITDB_ROOT_USERNAME=admin
MONGO_INITDB_ROOT_PASSWORD=secret

# Flask Configuration
FLASK_ENV=development
FLASK_DEBUG=1
SECRET_KEY=your_secure_secret_key

# MongoDB URI
MONGO_URI=mongodb://admin:secret@mongodb:27017/admin

# API Configuration
API_KEY=your_secure_api_key  # Replace with a strong, secure key

# LLM Integration (Future Use)
OPENAI_API_KEY=your_API_KEY  # Placeholder for future LLM integration
ENABLE_LLM=False

# Fuzzy Matching Configuration
FUZZY_MATCH_THRESHOLD=90       # Similarity threshold (0-100)
BATCH_SIZE=1000                # Number of entities to process per batch
FIELDS_TO_PROCESS=["ocr_text", "summary"]  # Fields to perform NER and fuzzy
matching on
ENABLE_MULTIPROCESSING=True    # Enable or disable multiprocessing in
processing scripts
LINK_WIKIDATA=False            # Enable or disable linking to Wikidata
```

Security Note: Ensure that `.env` is added to `.gitignore` to prevent accidental exposure of sensitive information.

---

2. Final Design Document Updates

2.1 Deployment and Infrastructure Enhancements

- **Health Checks for MongoDB:**
  - Implemented health checks in the Docker Compose configuration to ensure MongoDB is ready before the Flask application starts. This enhances reliability by preventing the Flask app from attempting to connect to an unavailable database.
- **Multi-Stage Dockerfile:**
  - Utilized a multi-stage Dockerfile to optimize the image size and separate build dependencies from the runtime environment. Pre-downloaded spaCy models in the build stage to expedite NER processing during runtime.
- **Entrypoint Script (`entrypoint.sh`):**
  - Introduced an entrypoint script to handle service initialization, including waiting for MongoDB to be ready. This script also allows for running setup scripts before launching the Flask application.
- **Volume Mounts for Initialization Scripts:**
  - Configured volume mounts for MongoDB initialization scripts (`./mongo-init/`) to automate the setup of databases, collections, and indexes upon container startup.

2.2 Security Considerations Enhancements

- **API Key Authentication:**
  - Added API key-based authentication for all API endpoints to secure communication between the local server and the HPC cluster.
- **CAPTCHA and Login Attempt Tracking:**
  - Enhanced the authentication mechanism in `routes.py` by implementing CAPTCHA verification and tracking login attempts to prevent brute-force attacks.
- **Environment Variable Management:**
  - Emphasized the use of environment variables for sensitive configurations, ensuring they are securely managed via the `.env` file and excluded from version control.

2.3 Monitoring and Logging Enhancements

- **Centralized Logging:**
  - Configured Docker logging drivers to manage log rotation, preventing log files from consuming excessive disk space. Logs for different services are segregated for better traceability.
- **Comprehensive Logging in Scripts:**
  - Enhanced logging in all scripts (`entity_linking.py`, `api_integration.py`, etc.) to facilitate easier debugging and monitoring of the processing workflows.

2.4 API Design Enhancements

- **Modular API Endpoints:**
  - Created additional API endpoints in `api_integration.py` to handle a variety of computational tasks, keeping the system abstracted and ready for future offloading requirements.
- **Compression and Data Transfer Optimization:**
  - Implemented data compression (gzip) for sending large payloads between the HPC cluster and the Flask API, optimizing bandwidth usage and improving transfer speeds.

2.5 Extensibility for Future Offloading Tasks

- **Abstracted Task Handling:**
  - Designed the API integration to handle different `task_type` values, allowing the system to support additional computational tasks beyond NER and fuzzy matching in the future.
- **Encapsulated Processing Logic:**
  - Structured processing scripts to be modular, making it easier to integrate new processing functionalities without disrupting existing workflows.

---

3. Additional Scripts and Components

3.1 Unique Terms Generation Script (`generate_unique_terms.py`)

This script generates a collection of unique terms from the documents for efficient fuzzy matching and analysis.

```
python
Copy code
# generate_unique_terms.py

import os
import logging
import json
from collections import defaultdict
from pymongo import UpdateOne, MongoClient
from dotenv import load_dotenv

# Load environment variables from .env file
load_dotenv()

# =======================
# Logging Configuration
# =======================
logger = logging.getLogger('UniqueTermsGenerator')
logger.setLevel(logging.DEBUG)

if not logger.handlers:
```

```
    console_handler = logging.StreamHandler()
    console_handler.setLevel(logging.INFO)

    file_handler = logging.FileHandler('unique_terms.log', mode='a')
    file_handler.setLevel(logging.DEBUG)

    formatter = logging.Formatter('%(asctime)s - %(levelname)s -
%(message)s')
    console_handler.setFormatter(formatter)
    file_handler.setFormatter(formatter)

    logger.addHandler(console_handler)
    logger.addHandler(file_handler)

# ======================
# Database Functions
# ======================

def get_client():
    """Initialize and return a new MongoDB client."""
    try:
        mongo_uri = os.environ.get('MONGO_URI')
        if not mongo_uri:
            raise ValueError("MONGO_URI environment variable not set")
        client = MongoClient(mongo_uri, serverSelectionTimeoutMS=5000)
        # Test connection
        client.admin.command('ping')
        logger.info("Successfully connected to MongoDB.")
        return client
    except Exception as e:
        logger.error(f"Failed to connect to MongoDB: {e}")
        raise e

def get_db(client):
    """Return the database instance."""
    return client['railroad_documents']

def get_collections(db):
    """Retrieve and return references to the required collections."""
    try:
        documents_collection = db['documents']
        unique_terms_collection = db['unique_terms']
        return documents_collection, unique_terms_collection
    except Exception as e:
        logger.error(f"Error getting collections: {e}")
        raise e

# ======================
# Unique Terms Generation
# ======================

def generate_unique_terms(documents_collection, unique_terms_collection,
fields_to_process, term_type='word', batch_size=1000):
    """
    Generate unique terms from specified fields in documents and update the
unique_terms collection.
```

```
    :param documents_collection: MongoDB collection containing documents
    :param unique_terms_collection: MongoDB collection to store unique terms
    :param fields_to_process: List of fields to extract terms from
    :param term_type: Type of term ('word' or 'phrase')
    :param batch_size: Number of documents to process per batch
    """
    logger.info(f"Starting unique terms generation for term_type:
{term_type}")
    cursor = documents_collection.find({}, {field: 1 for field in
fields_to_process})

    term_counts = defaultdict(int)

    for doc in cursor.batch_size(batch_size):
        for field in fields_to_process:
            text = doc.get(field, '')
            if text:
                if term_type == 'word':
                    words = text.split()
                    for word in words:
                        clean_word = word.strip().lower()
                        if clean_word:
                            term_counts[clean_word] += 1
                elif term_type == 'phrase':
                    # Example: Extract bigrams as phrases
                    words = text.split()
                    for i in range(len(words)-1):
                        phrase = f"{words[i].strip().lower()}
{words[i+1].strip().lower()}"
                        if phrase:
                            term_counts[phrase] += 1

    # Prepare bulk operations
    operations = []
    for term, count in term_counts.items():
        operations.append(
            UpdateOne(
                {"term": term, "type": term_type, "field": "ocr_text"},  #
Assuming 'ocr_text' is the field
                {"$set": {"term": term, "type": term_type, "field":
"ocr_text"}, "$inc": {"frequency": count}},
                upsert=True
            )
        )

    if operations:
        try:
            result = unique_terms_collection.bulk_write(operations,
ordered=False)
            logger.info(f"Bulk upserted {result.upserted_count +
result.modified_count} unique terms.")
        except Exception as e:
            logger.error(f"Error bulk upserting unique terms: {e}")
            raise e
    else:
        logger.info("No unique terms to upsert.")
```

```
    logger.info("Unique terms generation completed.")

# ======================
# Main Execution
# ======================

if __name__ == "__main__":
    try:
        client = get_client()
        db = get_db(client)
        documents_collection, unique_terms_collection = get_collections(db)

        # Define fields to process and term types
        fields_to_process = ["ocr_text", "summary"]
        term_types = ['word', 'phrase']

        for term_type in term_types:
            generate_unique_terms(
                documents_collection,
                unique_terms_collection,
                fields_to_process=fields_to_process,
                term_type=term_type,
                batch_size=1000
            )

    except Exception as e:
        logger.error(f"An error occurred during unique terms generation:
{e}", exc_info=True)
```

1.12 Docker Compose Configuration (`docker-compose.yml`)

Updated Docker Compose configuration incorporating health checks, multi-stage builds, and logging settings.

```
yaml
Copy code
version: '3.8'

services:
  mongodb:
    image: mongo:6.0
    container_name: mongodb
    environment:
      MONGO_INITDB_ROOT_USERNAME: admin
      MONGO_INITDB_ROOT_PASSWORD: secret
    volumes:
      - mongodb_data:/data/db
      - ./mongo-init/:/docker-entrypoint-initdb.d/
    ports:
      - "27017:27017"
    healthcheck:
      test: [
        "CMD",
        "mongosh",
```

```
        "--username",
        "admin",
        "--password",
        "secret",
        "--authenticationDatabase",
        "admin",
        "--eval",
        "db.adminCommand('ping')"
      ]
      interval: 10s
      timeout: 5s
      retries: 5
    networks:
      - app_network
    logging:
      driver: "json-file"
      options:
        max-size: "10m"
        max-file: "3"

  flask_app:
    build:
      context: ./app
      dockerfile: Dockerfile
    container_name: flask_app
    ports:
      - "5000:5000"
    environment:
      MONGO_URI:
"mongodb://${MONGO_INITDB_ROOT_USERNAME}:${MONGO_INITDB_ROOT_PASSWORD}@mongod
b:27017/admin"
      FLASK_APP: app.py
      FLASK_ENV: ${FLASK_ENV}
      FLASK_DEBUG: ${FLASK_DEBUG}
      SECRET_KEY: ${SECRET_KEY}
      API_KEY: ${API_KEY}
    volumes:
      - ./app:/app
      - ./archives:/app/archives
    depends_on:
      mongodb:
        condition: service_healthy
    networks:
      - app_network
    logging:
      driver: "json-file"
      options:
        max-size: "10m"
        max-file: "3"

volumes:
  mongodb_data:

networks:
  app_network:
    driver: bridge
```

```
# Notes:
# - Ensure you have a .env file in your project root with the necessary
variables.
# - Add .env to your .gitignore to prevent committing sensitive information.
```

Key Enhancements:

- **Health Checks:** Added health checks for MongoDB to ensure it's ready before the Flask app starts.
- **Logging:** Configured log rotation to prevent log files from growing indefinitely.
- **Environment Variables:** Included `API_KEY` in the Flask app's environment variables for secure API access.

---

3. Final Considerations for Extensibility

3.1 Abstracted Task Handling

The system is designed to handle various computational tasks beyond NER and fuzzy matching. By utilizing the `execute_task` endpoint in `api_integration.py`, additional processing scripts can be integrated seamlessly. Future tasks can be added by defining new `task_type` handlers within this script.

3.2 Encapsulation of Processing Logic

Processing scripts like `entity_linking.py`, `ner_processing.py`, and `generate_unique_terms.py` are modular and can be invoked independently or in conjunction with other scripts. This encapsulation ensures that adding new functionalities does not disrupt existing workflows.

3.3 Scalable HPC Integration

The HPC processing script (`hpc_processor.py`) is designed to fetch and execute tasks in batches, allowing for scalable processing as computational demands grow. Multiprocessing is enabled to utilize available CPU cores efficiently.

3.4 Secure and Efficient Communication

All communications between the HPC cluster and the local server are secured via API key authentication and can be further protected using HTTPS tunnels like ngrok. Data compression ensures efficient bandwidth usage, especially when dealing with large datasets.

---

4. Final Design Document Overview

The **Revised Design Document** incorporates the following key updates based on the newly provided Docker configurations and scripts:

- **Deployment and Infrastructure:**
  - Health checks for MongoDB.
  - Multi-stage Docker builds for optimized images.
  - Entrypoint scripts for service initialization.
  - Volume mounts for MongoDB initialization scripts.
- **Security Considerations:**
  - API key-based authentication for secure API access.
  - CAPTCHA and login attempt tracking to enhance security.
  - Secure management of environment variables.
- **Monitoring and Logging:**
  - Centralized and rotated logging configurations.
  - Comprehensive logging in all scripts for easier debugging.
- **API Design:**
  - Modular API endpoints to handle diverse computational tasks.
  - Data compression and efficient transfer mechanisms.
- **Extensibility:**
  - Abstracted and encapsulated processing logic to allow for future task offloading.
  - Scalable HPC integration with support for multiprocessing.
- **File Structure and Docker Setup:**
  - Clear and organized directory structure.
  - Docker Compose setup for easy deployment and management of services.

By implementing the above scripts and configurations, your system is now equipped to securely and efficiently offload NER and fuzzy matching tasks to an HPC cluster, with the flexibility to incorporate additional processing tasks in the future. Ensure that all environment variables are correctly set in the `.env` file and that sensitive information remains secured and excluded from version control.

If you have any further files or components to share, feel free to provide them, and I can assist in integrating them into the design and implementation.