Low-Level Design Document: Document Processing System

1. Introduction

This document details the low-level design of the document processing system, expanding upon the High-Level Design (HLD). It specifies the implementation details of each microservice, data structures, APIs, and communication mechanisms. We assume the requirements document ("output\extracted_docx.docx") specifies the data transformation and enrichment steps required by the Data Processing Service. This LLD uses Python with a PostgreSQL database as the technology stack, reflecting a common choice for this type of system. Alternatives are noted where appropriate.

2. Architecture Diagram

```plantuml
@startuml
!include <c4/C4_Context>
!include <c4/C4_Container>
!include <c4/C4_Component>

System_Boundary(c1, "Document Processing System") {
    Person(user, "Client Application", "Interacts with the system")
    Container(api_gateway, "API Gateway", "Kong", "Routes requests to microservices", "Kong")
    Container(document_ingestion, "Document Ingestion Service", "Python/Flask", "Receives, validates, and stores documents", "Python, Flask, AWS S3")
    Container(document_extraction, "Document Extraction Service", "Python", "Extracts data from documents", "Python, Apache Tika")
    Container(data_processing, "Data Processing Service", "Python", "Processes and transforms
extracted data", "Python, Pandas, scikit-learn (if NLP needed)")
        Container(data_storage, "Data Storage Service", "PostgreSQL", "Stores documents and processed data", "PostgreSQL")
    Container(monitoring, "Monitoring & Logging Service", "ELK Stack", "Collects and analyzes logs and metrics", "Elasticsearch, Logstash, Kibana")

   Rel(user, api_gateway, "Sends requests", "HTTP")
   Rel(api_gateway, document_ingestion, "Forwards requests", "HTTP")
   Rel(document_ingestion, document_extraction, "Sends document ID", "RabbitMQ")
   Rel(document_extraction, data_processing, "Sends extracted data", "RabbitMQ")
   Rel(data_processing, data_storage, "Stores processed data", "Database connection")
   Rel(document_ingestion, data_storage, "Stores documents", "AWS S3 API")
   Rel(document_ingestion, monitoring, "Sends logs and metrics", "Logstash")
   Rel(document_extraction, monitoring, "Sends logs and metrics", "Logstash")
   Rel(data_processing, monitoring, "Sends logs and metrics", "Logstash")
   Rel(data_storage, monitoring, "Sends logs and metrics", "Logstash")
   Rel(api_gateway, monitoring, "Sends logs and metrics", "Logstash")

}
@enduml
```

3. Data Flow

1. **Ingestion:** The client uploads a .docx file via a POST request to the API Gateway. The API Gateway forwards the request to the Document Ingestion Service. The service validates the file type, generates a UUID, stores the file in AWS S3, and sends the UUID and metadata (filename,

upload time) to the Message Queue (RabbitMQ).

2. **Extraction:** The Document Extraction Service consumes the message from the queue. It retrieves the document from AWS S3 using the UUID, extracts text, metadata (author, date, etc.), tables (as JSON), and images (stores image locations). The extracted data (in JSON format) is then sent to the Message Queue.

3. **Processing:** The Data Processing Service consumes the extracted data from the queue. It performs transformations specified in the requirements document (e.g., data cleaning, NLP tasks, data enrichment). The processed data (in JSON) is then sent to the Data Storage Service.

4. **Storage:** The Data Storage Service stores both the processed data and the document metadata in a PostgreSQL database. The processed data might be stored in JSONB columns for flexibility.

5. **Monitoring:** Each microservice sends logs and metrics to the Monitoring & Logging service (ELK Stack) for real-time monitoring and analysis.

4. Implementation Details

* **Document Ingestion Service:**
    * Uses Flask (or Django) for the HTTP API.
    * Uses the `boto3` library to interact with AWS S3.
    * Uses UUID library for unique document identifiers.
    * Uses RabbitMQ client library to publish messages.

* **Document Extraction Service:**

    * Uses Apache Tika library for document extraction.

    * Uses RabbitMQ client library to consume messages.

    * Uses a JSON library to serialize extracted data.

* **Data Processing Service:**

    * Uses Pandas for data manipulation and transformation.

    * Uses libraries like `NLTK` or `spaCy` for NLP tasks (if required).

    * Uses RabbitMQ client library to consume and publish messages.

* **Data Storage Service:**

    * Uses a PostgreSQL database with appropriate tables for documents and processed data.

    * Uses a database connector library (e.g., `psycopg2`).

* **API Gateway:**

    * Uses Kong API Gateway for routing and authentication.

* **Monitoring & Logging Service:**

    * Uses the ELK stack (Elasticsearch, Logstash, Kibana) for centralized logging and monitoring.

5. Class Diagrams (Example: Document Extraction Service)

```plantuml
@startuml

class DocumentExtractor {

    - documentId : String
    + extractData(documentId : String) : JSONObject
}

class DocumentRepository {
    + getDocument(documentId : String) : Document
}

class ApacheTikaWrapper {
    + extractText(document : Document) : String
    + extractMetadata(document : Document) : JSONObject
    + extractTables(document : Document) : JSONArray
}

DocumentExtractor *-- DocumentRepository
DocumentExtractor *-- ApacheTikaWrapper

@enduml
```

6. Sequence Diagrams (Example: Document Ingestion)

```plantuml
@startuml
actor Client
participant API Gateway
participant Document Ingestion Service
participant AWS S3
participant RabbitMQ


Client -> API Gateway : POST /documents

activate API Gateway

API Gateway -> Document Ingestion Service : Forward request

activate Document Ingestion Service

Document Ingestion Service -> AWS S3 : Upload document

activate AWS S3

AWS S3 --> Document Ingestion Service : Success/Failure

deactivate AWS S3

Document Ingestion Service -> RabbitMQ : Publish message (UUID, metadata)

deactivate Document Ingestion Service

API Gateway --> Client : Success/Failure

deactivate API Gateway

@enduml
```

This LLD provides a more detailed view of the system's implementation. Further refinement will be necessary based on the complete requirements document and ongoing development. Error handling, security considerations, and detailed database schema are further areas requiring detailed design.