TEAM 7

TEAM

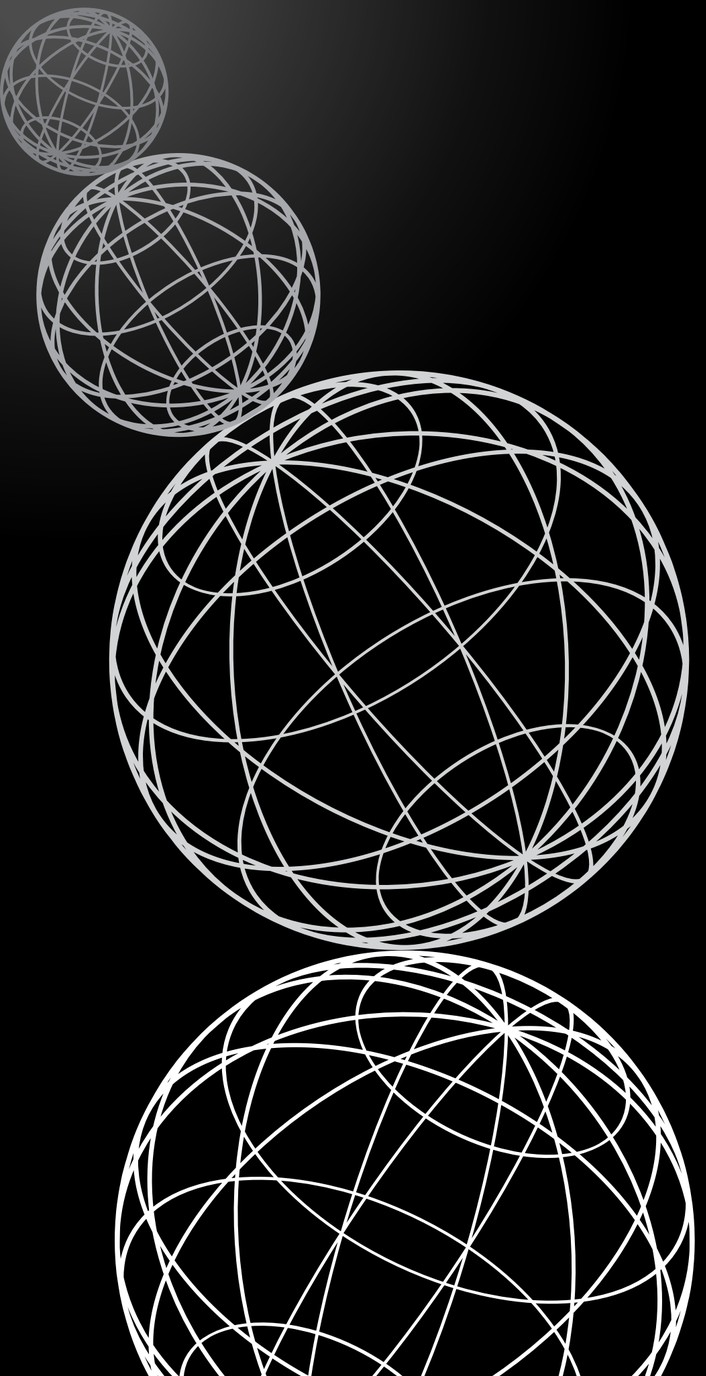
Team Member	Role	Coding Part
Diyorbek - 12230315	Frontend Designed the interface and layout	Frontend(index.html)
Akhrorbek - 12214752	Backend Built the API	Backend(app.py)
Elyor - 12214756	AI Integration Handled embeddings and LLM integration	Backend(rag.py)
Tilov - 12225275	Testing Ran tests and fixed bugs Built upload endpoint	Backend(app.py)
Abdulaziz - 12230330	Documentation Wrote reports and made the presentation Built document parser	Backend (parser.py)

PROBLEM STATEMENT

It's hard and time-consuming to go through long documents and find the exact information you need. Most search tools can't understand the context or meaning behind your question, so people end up wasting time reading everything manually.

SOLUTION OVERVIEW

We built a simple AI tool that lets users upload documents and ask questions in plain language. The tool reads the document, finds the most relevant parts, and gives clear answers or summaries. This helps users save time and understand content faster.



TEAM 7

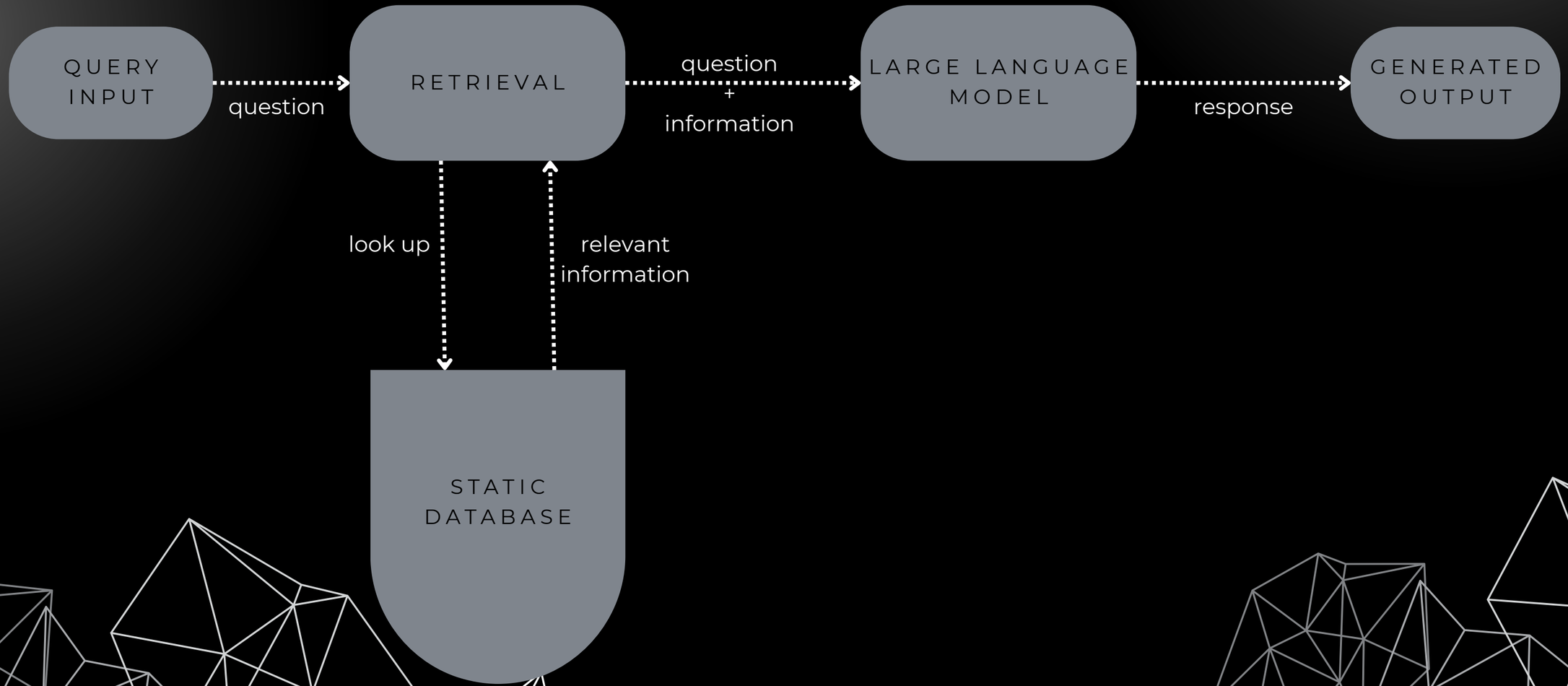
OBJECTIVE

To develop a lightweight, privacy-respecting document assistant that enables users to:

- Upload and manage documents (PDFs and text) via a clean REST API (/upload, /generate).
- Efficiently parse and extract semantic content from documents using SentenceTransformers.
- Store vector embeddings using FAISS for fast similarity search.
- Use local LLM inference to process and answer user queries.
- Implement a Retrieval-Augmented Generation (RAG) pipeline for accurate contextual responses.
- Build an interactive frontend that queries the REST API and displays intelligent results in real-time.

SYSTEM ARCHITECTURE

TEAM 7



BACKEND

PSEUDOCODE

RAG.PY

TEAM 7

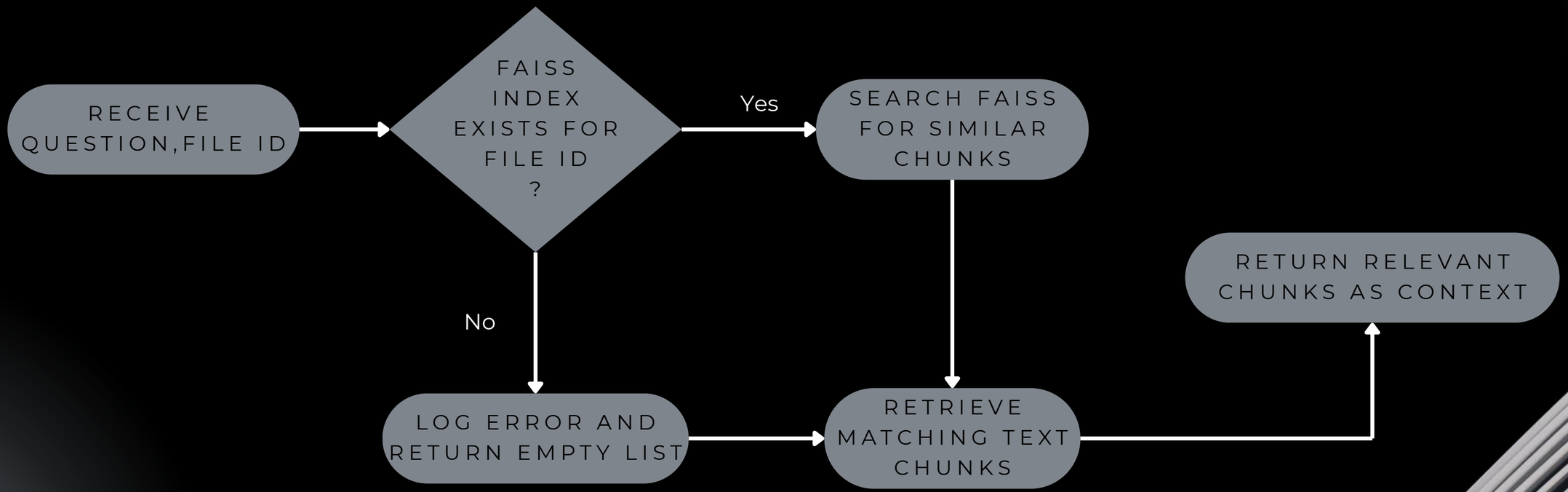
```
CLASS RAG:
    INIT:
        - Select device (GPU if available, else CPU)
        - Load SentenceTransformer model
        - Create FAISS index for vector search
        - Load existing index and metadata if
available
```

```
    FUNCTION embed_document(file_id, text):
        - Split text into sentences
        - Group sentences into chunks (~500
characters)
        - Add 3-sentence overlap between chunks
        - Generate embeddings for each chunk using
model
        - Normalize embeddings (for cosine similarity)
        - Add normalized embeddings to FAISS index
        - Save chunk metadata (file_id, text) for each
embedding
        - Persist index and metadata to disk
```

```
FUNCTION query_document(question, file_id):
        - Embed and normalize user query
        - Use FAISS to find top-k similar vectors
        - Filter results to only those matching file_id
        - Re-rank results using cosine similarity
        - Return top 3 most relevant chunks of text
```

ALGORITHM RAG.PY

TEAM 7



PSEUDOCODE

APP.PY

TEAM 7

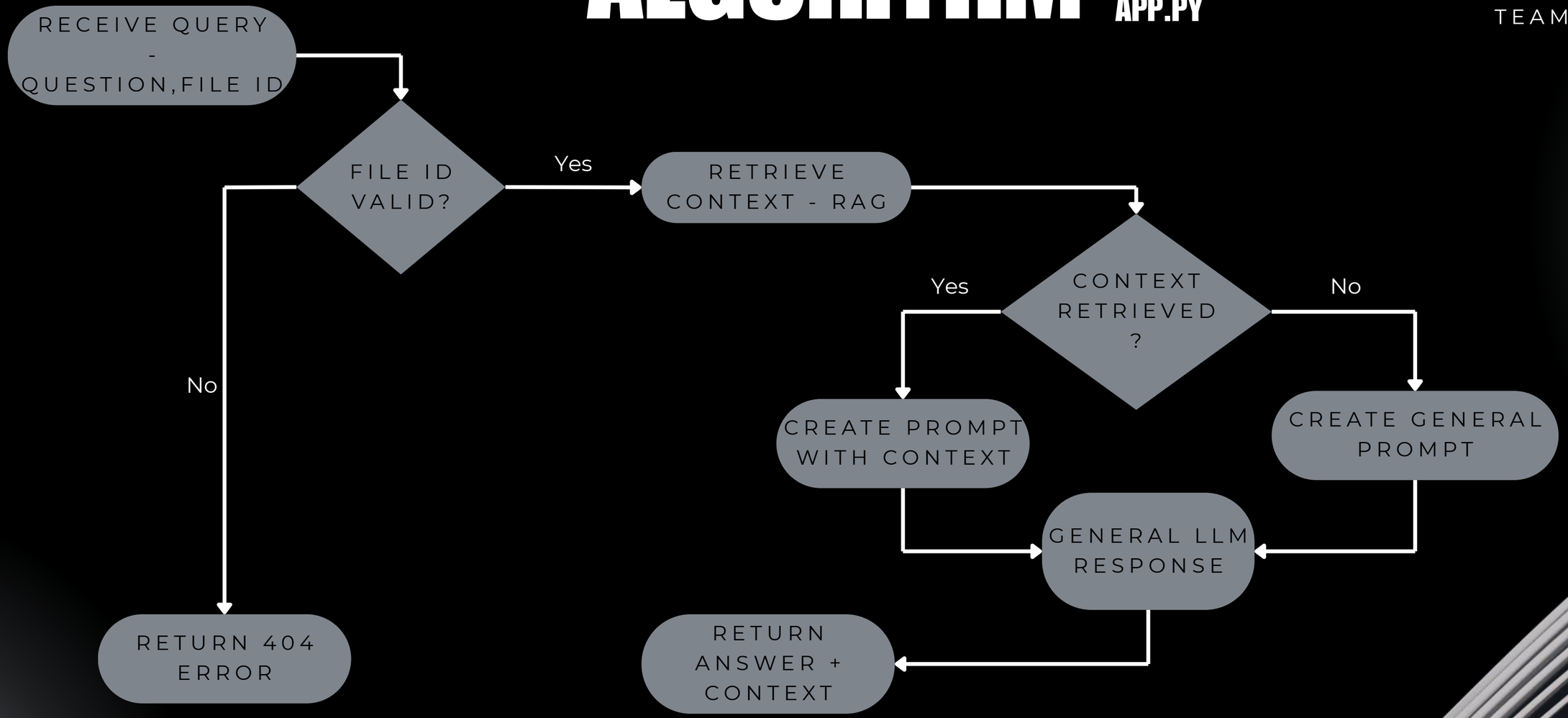
```
Log "Generating response from prompt"
TRY:
    Tokenize prompt using model's tokenizer
    Generate response using model
    Decode the output to text
    RETURN generated response
EXCEPT error:
    Log error message
    RETURN empty string
```

```
Log "Received query for file_id={request.file_id}"
TRY:
    IF request.file_id not in documents:
        Raise HTTPException with 404 (File not found)
    TRY:
        Retrieve context from RAG using request.question and request.file_id
    EXCEPT error:
        Log error message
        Set context as empty list
    IF context is empty:
        Log "No context found"
        Create a general prompt with request.question
    ELSE:
        Format prompt with context and request.question
```

```
TRY:
    Generate response using generate_response(prompt)
    RETURN response with generated answer and context
EXCEPT error:
    Log error message
    Raise HTTPException with 500 (Error generating answer)
```

ALGORITHM APP.PY

TEAM 7



PSEUDOCODE

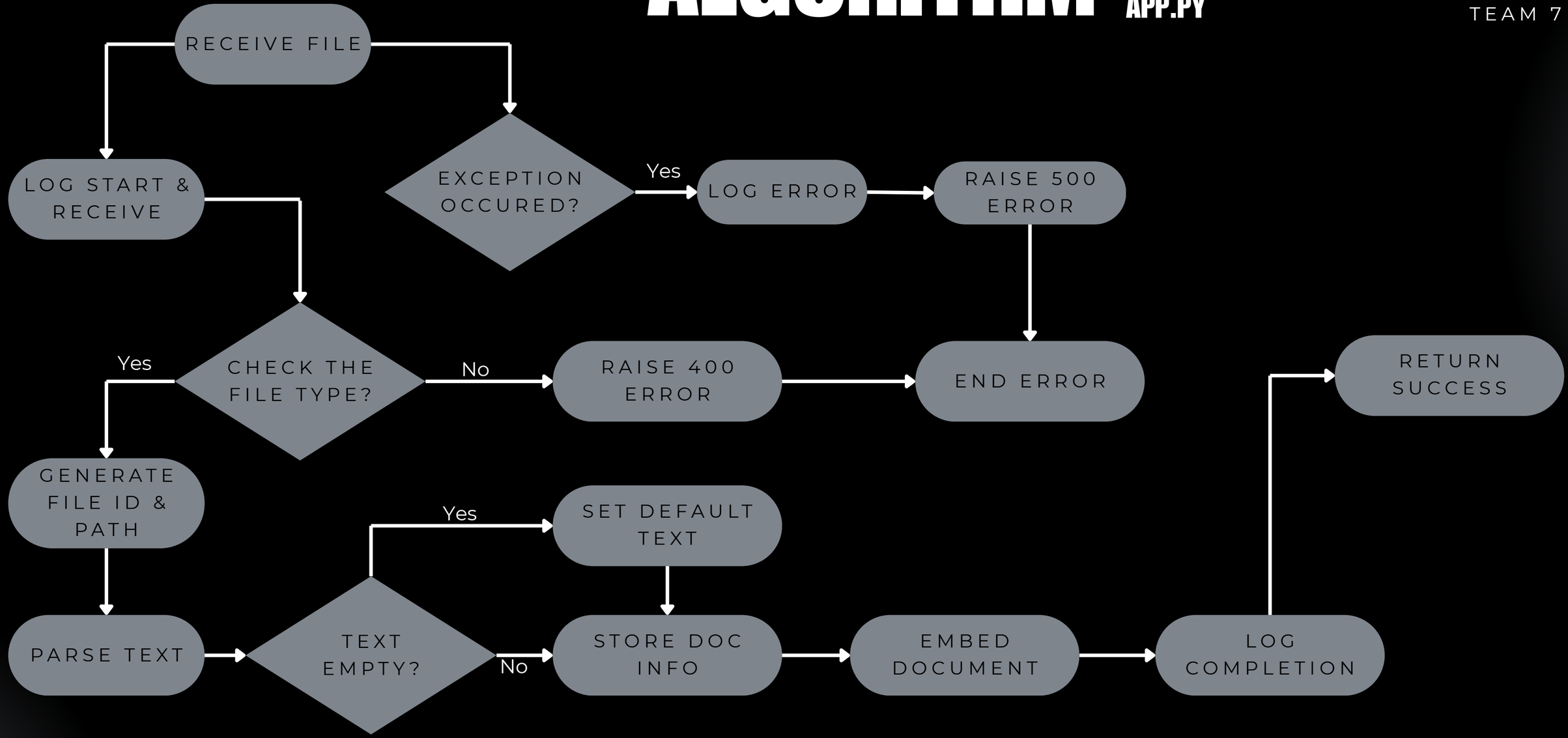
APP.PY

TEAM 7

```
FFUNCTION upload_file(file):
    LOG "Start upload"

    IF file type is not PDF or TXT:
        RETURN error "Invalid file type"

    CREATE unique file_id and safe filename
    ENSURE 'uploads' folder exists

    READ file content
    SAVE file to disk

    IF PDF:
        text = parse_pdf(file content)
    ELSE:
        text = parse_text(file content)

    IF text is empty:
        text = "No extractable text"

    STORE file metadata in memory (documents[file_id])

    CALL RAG.embed_document(file_id, text)

    LOG "Upload complete"
    RETURN file_id and file_name
```

ALGORITHM APP.PY

TEAM 7



PSEUDOCODE

PARSER.PY

```
FUNCTION parse_pdf(file_content):
    Log "Parsing PDF content"
    TRY:
        Convert file_content to a file-like object
        Create PDF reader from the file-like object
        Initialize empty string for extracted text
        FOR each page in the PDF:
            Extract text from the page
            IF text was extracted:
                Append it to the result text
        Log success
        RETURN extracted text
    EXCEPT error:
        Log error message
        RETURN empty string
```

```
FUNCTION parse_text(file_content):
    Log "Parsing text content"
    TRY:
        Decode file_content using UTF-8
        RETURN decoded text
    IF UnicodeDecodeError:
        TRY:
            Decode file_content using Latin-1
            RETURN decoded text
        EXCEPT error:
            Log error message
            RETURN empty string
```

ALGORITHM PARSER.PY

TEAM 7

FRONTEND

PSEUDOCODE

INDEX.HTML

```
# Initialize server and frontend
Initialize web_server
Initialize frontend_interface

# File upload handler
def upload_document(file):
    if valid_file_type(file):
        document_id = save_file(file)
        text_content = extract_text(file)
        store_document_data(document_id,
text_content)
        return success("File uploaded", document_id)
    return error("Invalid file type")
```

```
# User query handler
def handle_query(document_id, user_query):
    document = retrieve_document_data(document_id)
    ai_response = generate_answer(document, user_query)
    return ai_response

# Frontend actions
function on_file_upload(event):
    file = event.target.files[0]
    send_file_to_server(file)

function on_user_query(event):
    query = event.target.value
    ai_response = send_query_to_server(query)
    display_response(ai_response)
```

```
# Server setup
def run_server():
    web_server.add_endpoint("/upload", upload_document, method="POST")
    web_server.add_endpoint("/query", handle_query, method="POST")
    web_server.start()

# Start web interface
function start_interface():
    setup_file_upload_ui()
    setup_chat_ui()
    listen_for_user_interactions()

run_server()
start_interface()
```

CHALLENGES & SOLUTIONS

Hardware Incompatibility

Switched to optimized quantized models (e.g., gguf versions via llama.cpp) for local inference.
Used batch processing and offloaded memory with 4-bit precision.

Communication Issues

Used OpenAPI/Swagger docs for shared API understanding.
Scheduled short sync meetings and used Postman for testing endpoints.

File Parsing Edge Cases

Added robust error handling.
Used pdfplumber and python-docx with fallback strategies and logging.

LLM Response Irrelevance

Strict context formatting.
Tested multiple prompt styles and enforced grounding via [CONTEXT] delimiters.

FUTURE ENHANCEMENTS

- Add support for images and scanned PDFs using OCR
- Add citation tracking for answers
- Improve summarization quality with larger LLMs
- Mobile responsive UI and Support voice input
- Add user accounts and session history

THANK YOU

TEAM 7