Penjelasan Detail Kode Sistem Tiket RAG - Function by Function

1. IMPORT LIBRARIES DAN PENGGUNAAN

Standard Python Libraries

```python
import os
import json
import logging
from datetime import datetime
```

- **os**: `os.getenv()` - untuk membaca environment variables
- **json**: `json.dumps()` - untuk serialisasi data Python ke JSON string
- **logging**: `logging.basicConfig()`, `logger.info()`, `logger.error()` - sistem pencatatan log
- **datetime**: `datetime` - untuk timestamp (meskipun tidak digunakan langsung dalam kode ini)

Database Libraries

```python
import psycopg2
from psycopg2.extras import RealDictCursor
```

- **psycopg2**:
  - `psycopg2.connect()` - membuat koneksi ke PostgreSQL
  - `psycopg2.Error` - exception handling untuk database errors
- **RealDictCursor**: Cursor yang mengembalikan hasil query sebagai dictionary

LangChain Libraries

```python
```

```python
from langchain_google_genai import ChatGoogleGenerativeAI, GoogleGenerativeAIEmbeddings
from langchain_community.utilities import SQLDatabase
from langchain_core.documents import Document
from langchain_postgres import PGVector
from langchain_core.prompts import ChatPromptTemplate
from langchain.chains import create_retrieval_chain
from langchain.chains.combine_documents import create_stuff_documents_chain
from langchain_core.output_parsers import JsonOutputParser
```

- **ChatGoogleGenerativeAI**: Interface untuk Google Gemini model
- **GoogleGenerativeAIEmbeddings**: Model embedding dari Google untuk konversi teks ke vektor
- **SQLDatabase**: Wrapper untuk operasi database SQL
- **Document**: Class untuk menyimpan dokumen dengan content dan metadata
- **PGVector**: Vector store implementation menggunakan PostgreSQL
- **ChatPromptTemplate**: Template untuk membuat prompt chat
- **create_retrieval_chain**: Membuat chain untuk retrieval + generation
- **create_stuff_documents_chain**: Chain untuk menggabungkan dokumen ke dalam prompt
- **JsonOutputParser**: Parser untuk output JSON

Validation & Environment

```python
from pydantic import BaseModel, Field
from dotenv import load_dotenv
from sqlalchemy import text
```

- **BaseModel**: Base class untuk model Pydantic
- **Field**: Untuk mendefinisikan field dengan deskripsi dan validasi
- **load_dotenv**: Memuat environment variables dari file .env
- **text**: SQLAlchemy function untuk raw SQL queries

2. KONFIGURASI & LOGGING

```python
```

```
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler('D:\\interface\\ticketing_system_cron.log'),
        logging.StreamHandler()
    ]
)
logger = logging.getLogger(__name__)
```

Functions Used:

- `logging.basicConfig()`: Konfigurasi dasar logging
- `logging.FileHandler()`: Handler untuk menulis log ke file
- `logging.StreamHandler()`: Handler untuk output ke console
- `logging.getLogger()`: Mendapatkan logger instance

3. PYDANTIC MODEL

```python
class TicketAnalysis(BaseModel):
    issue: str = Field(description="Salin ulang pertanyaan/keluhan dari pengguna")
    priority: str = Field(description="Pilih salah satu: P1 (Kritis), P2 (Tinggi), P3 (Sedang), P4 (Rendah)")
    unit: str = Field(description="Rekomendasikan tim atau departemen yang paling sesuai")
    solution: str = Field(description="Solusi awal yang dapat membantu pengguna")
    justification: str = Field(description="Justifikasi kenapa keluhan tersebut memiliki prioritas tertentu")
```

Functions/Methods Used:

- `BaseModel`: Pydantic base class untuk data validation
- `Field()`: Definisi field dengan description untuk AI parsing

4. DATABASE FUNCTIONS

get_db_connection()

```python
```

```python
def get_db_connection():
    try:
        connection = psycopg2.connect(
            host=DB_HOST,
            database=DB_NAME,
            user=DB_USER,
            password=DB_PASSWORD,
            port=DB_PORT
        )
        return connection
    except psycopg2.Error as e:
        logger.error(f"Database connection error: {e}")
        return None
```

Functions Used:

- `psycopg2.connect()`: Membuat koneksi PostgreSQL dengan parameter host, database, user, password, port
- `logger.error()`: Mencatat error ke log

update_ticket_status()

```
python
```

```
def update_ticket_status(ticket_id, new_status):
    conn = None
    try:
        conn = get_db_connection()
        if conn is None:
            return False

        with conn.cursor() as cur:
            cur.execute("""
                UPDATE data.rag_queries
                SET status = %s
                WHERE id = %s
            """, (new_status, ticket_id))
            conn.commit()
            logger.info(f"Ticket {ticket_id} status updated to {new_status}")
            return True

    except psycopg2.Error as e:
        logger.error(f"Error updating ticket status for ticket {ticket_id}: {e}")
        if conn:
            conn.rollback()
        return False
    finally:
        if conn:
            conn.close()
```

Functions Used:

- `conn.cursor()`: Membuat cursor untuk eksekusi SQL
- `cur.execute()`: Eksekusi SQL query dengan parameterized values
- `conn.commit()`: Commit transaksi ke database
- `conn.rollback()`: Rollback transaksi jika error
- `conn.close()`: Tutup koneksi database
- `logger.info()` dan `logger.error()`: Logging

save_rag_result()

```
python
```

```
def save_rag_result(ticket_id, issue, solution, sources):
    conn = None
    try:
        conn = get_db_connection()
        if conn is None:
            return False

        with conn.cursor() as cur:
            cur.execute("""
                INSERT INTO data.rag_logs (query_id, issue, solution, sources)
                VALUES (%s, %s, %s, %s)
            """, (ticket_id, issue, json.dumps(solution, ensure_ascii=False), json.dumps(sources, ensure_ascii=False)))
            conn.commit()
            logger.info(f"RAG result saved for ticket {ticket_id}")
            return True
    # ... error handling sama seperti update_ticket_status
```

Functions Used:

- `json.dumps()`: Serialize Python object ke JSON string dengan `ensure_ascii=False` untuk UTF-8 encoding
- Fungsi database sama seperti `update_ticket_status()`

5. RAG SYSTEM INITIALIZATION

initialize_rag_system()

Model Initialization

```python
chat_model = ChatGoogleGenerativeAI(
    google_api_key=GOOGLE_API_KEY,
    model='gemini-2.5-pro',
    temperature=0.3
)

embeddings = GoogleGenerativeAIEmbeddings(
    model="models/embedding-001",
    google_api_key=GOOGLE_API_KEY
)
```

Functions/Classes Used:

- **ChatGoogleGenerativeAI()**: Constructor untuk chat model dengan parameter API key, model name, dan temperature
- **GoogleGenerativeAIEmbeddings()**: Constructor untuk embedding model

Database Setup

```python
db = SQLDatabase.from_uri(DB_URI, schema="data")

def load_documents_from_db():
    with db._engine.connect() as conn:
        try:
            result = conn.execute(text("""
                SELECT DISTINCT "ID Tiket", "Keluhan", "Prioritas", "Justifikasi Prioritas", "Unit Penanggung Jawab", "Solusi Aw
                FROM data.dataset_dummy_ticketing
                WHERE "Keluhan" IS NOT NULL AND "Prioritas" IS NOT NULL;
            """))

            rows = result.fetchall()
            docs = [
                Document(
                    page_content=row[1],
                    metadata={
                        "ticket_id": row[0], "prioritas": row[2], "justifikasi_prioritas": row[3] or "",
                        "unit_penanggung_jawab": row[4] or "", "solusi_awal": row[5] or ""
                    }
                ) for row in rows
            ]
            logger.info(f"Loaded {len(docs)} documents from database")
            return docs
```

Functions Used:

- **SQLDatabase.from_uri()**: Membuat SQLDatabase object dari URI
- **db._engine.connect()**: Mendapatkan connection dari SQLAlchemy engine
- **conn.execute(text())**: Eksekusi raw SQL query
- **result.fetchall()**: Ambil semua rows dari hasil query
- **Document()**: Constructor untuk LangChain Document dengan page_content dan metadata
- **len()**: Hitung jumlah dokumen

Vector Store Setup

```python
vector_store = PGVector(
    embeddings=embeddings,
    collection_name=COLLECTION_NAME,
    connection=VECTOR_DB_CONNECTION,
    use_jsonb=True,
)

docs = load_documents_from_db()
if docs:
    try:
        vector_store.delete_collection()
        vector_store.create_collection()

        batch_size = 100
        for i in range(0, len(docs), batch_size):
            batch = docs[i:i + batch_size]
            vector_store.add_documents(batch)
        logger.info(f"Successfully added {len(docs)} documents to vector store")
```

Functions/Methods Used:

- **`PGVector()`**: Constructor untuk PostgreSQL vector store
- **`vector_store.delete_collection()`**: Hapus collection yang ada
- **`vector_store.create_collection()`**: Buat collection baru
- **`range()`**: Generate range numbers untuk batch processing
- **`vector_store.add_documents()`**: Tambahkan batch dokumen ke vector store

Chain Creation

```python
```

```
retriever = vector_store.as_retriever(search_kwargs={"k": 5})
parser = JsonOutputParser(pydantic_object=TicketAnalysis)

system_prompt = (
    "Anda adalah asisten AI yang sangat efisien untuk tim dukungan IT..."
)

prompt = ChatPromptTemplate.from_messages([
    ("system", system_prompt),
    ("human", "{input}"),
])

prompt = prompt.partial(format_instructions=parser.get_format_instructions())

question_answer_chain = create_stuff_documents_chain(chat_model, prompt)
rag_chain = create_retrieval_chain(retriever, question_answer_chain)
```

Functions/Methods Used:

- **`vector_store.as_retriever()`**: Convert vector store ke retriever dengan k=5 (ambil 5 dokumen terdekat)
- **`JsonOutputParser()`**: Parser dengan Pydantic model
- **`ChatPromptTemplate.from_messages()`**: Buat prompt template dari list messages
- **`prompt.partial()`**: Partial prompt dengan format instructions
- **`parser.get_format_instructions()`**: Dapatkan instruksi format JSON
- **`create_stuff_documents_chain()`**: Chain yang menggabungkan dokumen ke prompt
- **`create_retrieval_chain()`**: Chain yang menggabungkan retrieval + generation

Output Parser Function

```
python
```

```python
def parse_rag_output(inputs):
    result = None
    try:
        result = rag_chain.invoke(inputs)
        parsed_answer = parser.parse(result['answer'])

        if not isinstance(parsed_answer, dict):
            raise ValueError("Parsed result is not a dictionary")

        required_fields = ['issue', 'priority', 'unit', 'solution', 'justification']
        for field in required_fields:
            if field not in parsed_answer or not parsed_answer[field]:
                parsed_answer[field] = 'Tidak Diketahui'

        valid_priorities = ['P1 (Kritis)', 'P2 (Tinggi)', 'P3 (Sedang)', 'P4 (Rendah)']
        if parsed_answer['priority'] not in valid_priorities:
            parsed_answer['priority'] = 'P3 (Sedang)'

        return {'answer': parsed_answer, 'context': result['context']}
```

Functions Used:

- `rag_chain.invoke()`: Execute RAG chain dengan input
- `parser.parse()`: Parse string JSON ke Python object
- `isinstance()`: Check type dari object
- `raise ValueError()`: Raise exception jika ada error
- **Dictionary operations**: `in`, key access, assignment untuk validation
- `.get()`: Safe dictionary access dengan default value

6. PROCESSING FUNCTIONS

process_single_ticket()

```
python
```

```python
def process_single_ticket(ticket_id, question, rag_chain_parser):
    try:
        logger.info(f"Processing ticket {ticket_id}: {question[:50]}...")

        if not update_ticket_status(ticket_id, 'processing'):
            return False, "Failed to update ticket status to 'processing'"

        result = rag_chain_parser({"input": question})
        parsed_answer = result['answer']

        if 'parse_error' in result:
             raise Exception(f"Parsing failed: {result['parse_error']}")

        sources = [{"content": doc.page_content, "metadata": doc.metadata} for doc in result['context']]

        if save_rag_result(ticket_id, question, parsed_answer, sources):
            if update_ticket_status(ticket_id, 'done'):
                logger.info(f"Successfully processed ticket {ticket_id}")
                return True, parsed_answer
            else:
                logger.error(f"Failed to update ticket {ticket_id} status to 'done'")
                return False, "Failed to update ticket status to 'done'"
```

Functions Used:

- **String slicing**: `question[:50]` - ambil 50 karakter pertama
- **Function calls**: `update_ticket_status()`, `save_rag_result()`
- **List comprehension**: Extract content dan metadata dari context documents
- **Dictionary access**: `result['answer']`, `result['context']`
- **`raise Exception()`**: Raise custom exception

process_tickets_batch()

```
python
```

```python
def process_tickets_batch(rag_chain_parser, max_tickets=5):
    try:
        conn = get_db_connection()
        if conn is None:
            return {'error': 'Database connection failed'}

        with conn.cursor(cursor_factory=RealDictCursor) as cur:
            cur.execute("""
                SELECT id, question FROM data.rag_queries
                WHERE status = 'pending' ORDER BY id ASC LIMIT %s
            """, (max_tickets,))
            pending_tickets = cur.fetchall()

        conn.close()

        if not pending_tickets:
            return {'message': 'No pending tickets found'}

        results = []
        for ticket in pending_tickets:
            success, result_data = process_single_ticket(
                ticket['id'], ticket['question'], rag_chain_parser
            )
            results.append({
                'ticket_id': ticket['id'],
                'success': success,
                'result': result_data if success else f"Error: {result_data}"
            })
```

Functions Used:

- **`conn.cursor(cursor_factory=RealDictCursor)`**: Cursor yang return dictionary
- **SQL with parameters**: Parameterized query dengan LIMIT
- **`cur.fetchall()`**: Fetch semua hasil query
- **`conn.close()`**: Tutup koneksi
- **List operations**: `results.append()`, iteration dengan `for`
- **Conditional expressions**: Ternary operator `if success else`
- **Dictionary access**: `ticket['id']`, `ticket['question']`

7. MAIN EXECUTION

run_job()

```python
def run_job():
    logger.info("Cron job started: Processing pending tickets.")

    rag_chain_parser, db = initialize_rag_system()
    if not rag_chain_parser:
        logger.error("Failed to initialize RAG system. Aborting job.")
        return

    try:
        max_tickets_to_process = 10
        result = process_tickets_batch(rag_chain_parser, max_tickets=max_tickets_to_process)

        if 'error' in result:
            logger.error(f"Batch processing failed with error: {result['error']}")
        elif 'message' in result:
            logger.info(f"Batch processing status: {result['message']}")
        elif 'results' in result:
            processed_count = len(result['results'])
            success_count = sum(1 for r in result['results'] if r['success'])
            logger.info(f"Batch processing finished. Processed: {processed_count}, Successful: {success_count}.")
            for res in result['results']:
                if not res['success']:
                    logger.warning(f"Failed ticket ID {res['ticket_id']}: {res['result']}")

    except Exception as e:
        logger.error(f"An unexpected error occurred during the job: {e}", exc_info=True)

    logger.info("Cron job finished.")
```

Functions Used:

- **Multiple assignment**: `rag_chain_parser, db = initialize_rag_system()`
- **Dictionary membership**: `'error' in result`, `'message' in result`
- **`len()`**: Hitung jumlah hasil
- **`sum()` dengan generator**: Count success dengan kondisi
- **`logger.warning()`**: Log level warning
- **Exception handling**: `exc_info=True` untuk full stack trace

Entry Point

```python
if __name__ == "__main__":
    run_job()
```

Python Built-in:

- `__name__`: Special variable berisi nama module
- `__main__`: Value saat script dijalankan langsung

SUMMARY FUNCTIONS PER LIBRARY

psycopg2 Functions:

- `psycopg2.connect()` - koneksi database
- `conn.cursor()` - buat cursor
- `cur.execute()` - eksekusi query
- `conn.commit()` - commit transaksi
- `conn.rollback()` - rollback transaksi
- `conn.close()` - tutup koneksi
- `RealDictCursor` - cursor yang return dict

LangChain Functions:

- `ChatGoogleGenerativeAI()` - chat model
- `GoogleGenerativeAIEmbeddings()` - embedding model
- `SQLDatabase.from_uri()` - database wrapper
- `Document()` - document object
- `PGVector()` - vector store
- `ChatPromptTemplate.from_messages()` - prompt template
- `JsonOutputParser()` - JSON parser
- `create_retrieval_chain()` - RAG chain
- `create_stuff_documents_chain()` - document chain

Python Standard Library:

- `logging.*` - sistem logging

- `json.dumps()` - JSON serialization
- `os.getenv()` - environment variables
- List comprehensions, dictionary operations, string operations
- Exception handling dengan try/except/finally

Pydantic Functions:

- `BaseModel` - base class
- `Field()` - field definition
- Data validation dan parsing otomatis