项目申请书

项目名称：openGauss 向量数据库 AI 数据 ETL 框架最佳实践输出

项目编号：2522d0324

项目导师：mpb159753

申请人：曾繁曦

申请人邮箱：2628690042@qq.com

一、项目背景

1.1项目简述

完成 Apache Kafka、Firecrawl、VectorETL 对接 openGauss 最佳实践教程输出

1.2 项目产出要求

1. 基于 openGauss 部署指导文档，完成openGauss Docker 部署
2. 参考 Milvus weaviate 等文档，完成 Apache Kafka、Firecrawl、VectorETL 对接 openGauss 最佳实践教程输出
3. 将文档提交至 openGauss 社区

1.3 项目技术要求

1. 了解docker构建部署
2. 了解openGauss的基础功能
3. 了解Python编程开发
4. 了解RAG/LLMs基础知识

1.4 项目成果仓库

- https://gitcode.com/opengauss/examples

二、Apache Kafka 对接 openGauss 实现方案

1. 架构设计

Apache Kafka 与 openGauss 的集成主要通过以下两种模式实现：

1.1 实时数据管道模式

```
[生产者应用] → [Kafka Topic] → [Kafka Connect] → [openGauss]
```

1.2 自定义消费者模式

```
[生产者应用] → [Kafka Topic] → [自定义消费者应用] → [openGauss]
```

2. 前置准备

2.1 环境要求

- Apache Kafka 2.8+ 集群
- openGauss 3.0+ 数据库（已安装vector扩展）
- JDK 11+ 环境
- 网络互通性验证

2.2 依赖组件

- Kafka Connect 框架
- openGauss JDBC 驱动（opengauss-jdbc-5.0.0.jar）
- Confluent JDBC Sink Connector 或 Debezium Connector

3. 核心实现方案

3.1 方案一：使用Kafka Connect JDBC Sink

3.1.1 安装配置步骤

1. **部署JDBC连接器**：

```
# 下载Confluent JDBC连接器
wget https://packages.confluent.io/maven/io/confluent/kafka-connect-jdbc/10.7.0/kafka-connect-jdbc-10.7.0.tar.gz
tar -xzf kafka-connect-jdbc-10.7.0.tar.gz -C /usr/share/java/
```

2. **配置openGauss驱动：**

```
代码块
1   cp opengauss-jdbc-5.0.0.jar /usr/share/java/kafka-connect-jdbc/
```

3. **创建Sink配置（jdbc-sink-opengauss.properties）：**

```
代码块
1   name=opengauss-sink
2   connector.class=io.confluent.connect.jdbc.JdbcSinkConnector
3   tasks.max=3
4   topics=source_topic
5   connection.url=jdbc:opengauss://opengauss_host:5432/target_db
6   connection.user=kafka_user
7   connection.password=db_password
8   auto.create=false
9   auto.evolve=true
10  insert.mode=upsert
11  pk.mode=record_key
12  pk.fields=id
13  table.name.format=target_table
14  transforms=unwrap
15  transforms.unwrap.type=io.debezium.transforms.ExtractNewRecordState
```

4. **启动连接器：**

```
代码块
1   curl -X POST -H "Content-Type: application/json" \
2     --data @jdbc-sink-opengauss.properties \
3     http://kafka-connect:8083/connectors
```

3.2 方案二：自定义Java消费者应用

3.2.1 Java实现代码

```
代码块
1   import org.apache.kafka.clients.consumer.*;
2   import org.apache.kafka.common.serialization.StringDeserializer;
3   import java.sql.*;
4   import java.time.Duration;
5   import java.util.Collections;
```

```
import java.util.Properties;

public class OpenGaussKafkaConsumer {
    private static final String TOPIC = "vector_data";
    private static final String BOOTSTRAP_SERVERS = "kafka1:9092,kafka2:9092";
    private static final String GROUP_ID = "opengauss-consumer-group";
    private static final String OPENGAUSS_URL =
"jdbc:opengauss://opengauss:5432/vector_db";

    public static void main(String[] args) {
        // Kafka消费者配置
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
        props.put(ConsumerConfig.GROUP_ID_CONFIG, GROUP_ID);
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "false");

        // openGauss连接池配置
        PGSimpleDataSource ds = new PGSimpleDataSource();
        ds.setURL(OPENGAUSS_URL);
        ds.setUser("kafka_user");
        ds.setPassword("db_password");

        try (KafkaConsumer<String, String> consumer = new KafkaConsumer<>
(props)) {
            consumer.subscribe(Collections.singleton(TOPIC));

            while (true) {
                ConsumerRecords<String, String> records =
consumer.poll(Duration.ofMillis(100));

                try (Connection conn = ds.getConnection()) {
                    conn.setAutoCommit(false);

                    for (ConsumerRecord<String, String> record : records) {
                        processRecord(conn, record);
                    }

                    conn.commit();
                    consumer.commitSync();
                } catch (SQLException e) {
                    System.err.println("Database error: " + e.getMessage());
                }
```

```
            }
        }
    }

    private static void processRecord(Connection conn, ConsumerRecord<String,
String> record)
            throws SQLException {
        // 解析JSON数据（示例使用简单字符串，实际可用Jackson/Gson）
        String[] parts = record.value().split("\\|");

        try (PreparedStatement pstmt = conn.prepareStatement(
                "INSERT INTO vector_messages (id, content, vector) " +
                "VALUES (?, ?, ?::vector) " +
                "ON CONFLICT (id) DO UPDATE SET " +
                "content = EXCLUDED.content, " +
                "vector = EXCLUDED.vector")) {

            pstmt.setString(1, record.key());
            pstmt.setString(2, parts[0]);
            pstmt.setObject(3, conn.createArrayOf("float",
parseVector(parts[1])));
            pstmt.executeUpdate();
        }
    }

    private static Float[] parseVector(String vectorStr) {
        String[] nums = vectorStr.split(",");
        Float[] vector = new Float[nums.length];
        for (int i = 0; i < nums.length; i++) {
            vector[i] = Float.parseFloat(nums[i].trim());
        }
        return vector;
    }
}
```

4. 高级配置与优化

4.1 批量写入优化

代码块

```
# 在连接器配置中添加
batch.size=5000
max.retries=3
retry.backoff.ms=1000
```

4.2 向量数据处理

对于openGauss的向量类型，需要特殊处理：

代码块

```
-- 创建带向量字段的表
CREATE TABLE vector_data (
    id VARCHAR(128) PRIMARY KEY,
    content TEXT,
    embedding VECTOR(384),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

在消费者中处理向量数据：

代码块

```
// 在processRecord方法中使用向量参数
pstmt.setObject(3, conn.createArrayOf("float", vectorArray));
```

5. 完整部署示例（Docker Compose）

代码块

```
version: '3.8'

services:
  zookeeper:
    image: confluentinc/cp-zookeeper:7.0.0
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181

  kafka:
    image: confluentinc/cp-kafka:7.0.0
    depends_on:
      - zookeeper
    ports:
      - "9092:9092"
    environment:
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:9092
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1

  kafka-connect:
    image: confluentinc/cp-kafka-connect:7.0.0
```

```
    depends_on:
      - kafka
    ports:
      - "8083:8083"
    environment:
      CONNECT_BOOTSTRAP_SERVERS: kafka:9092
      CONNECT_REST_ADVERTISED_HOST_NAME: kafka-connect
      CONNECT_GROUP_ID: connect-cluster
      CONNECT_CONFIG_STORAGE_TOPIC: connect-configs
      CONNECT_OFFSET_STORAGE_TOPIC: connect-offsets
      CONNECT_STATUS_STORAGE_TOPIC: connect-status
      CONNECT_KEY_CONVERTER: org.apache.kafka.connect.json.JsonConverter
      CONNECT_VALUE_CONVERTER: org.apache.kafka.connect.json.JsonConverter
      CONNECT_CONFIG_STORAGE_REPLICATION_FACTOR: 1
      CONNECT_OFFSET_STORAGE_REPLICATION_FACTOR: 1
      CONNECT_STATUS_STORAGE_REPLICATION_FACTOR: 1
      CONNECT_PLUGIN_PATH: "/usr/share/java,/usr/share/confluent-hub-
components"
    volumes:
      - ./opengauss-jdbc-5.0.0.jar:/usr/share/java/kafka-connect-
jdbc/opengauss-jdbc-5.0.0.jar
      - ./jdbc-sink-opengauss.properties:/tmp/jdbc-sink-opengauss.properties

  opengauss:
    image: enmotech/opengauss:3.0.0
    environment:
      GS_PASSWORD: "OpenGauss@123"
    ports:
      - "5432:5432"
    volumes:
      - opengauss_data:/var/lib/opengauss

volumes:
  opengauss_data:
```

启动后初始化连接器：

代码块

```
docker-compose exec kafka-connect \
  curl -X POST -H "Content-Type: application/json" \
  --data @/tmp/jdbc-sink-opengauss.properties \
  http://localhost:8083/connectors
```

三、Firecrawl 对接 openGauss 实现方案

1. 整体架构设计

Firecrawl (如 Firecrawl API 或开源爬虫框架) 与 openGauss 的集成主要通过以下流程实现：

```
代码块
1  [Firecrawl 爬取数据] → [数据预处理] → [向量化处理] → [存储到 openGauss]
```

2. 前置准备

2.1 环境要求

- 已部署的 Firecrawl 服务或 API 访问权限
- openGauss 3.0+ 数据库(已启用向量扩展)
- Python 3.8+ 环境
- 网络连通性(Firecrawl 可访问目标网站，应用服务器可访问 openGauss)

2.2 依赖安装

```
代码块
1  pip install opengauss-python psycopg2-binary requests beautifulsoup4 numpy
   sentence-transformers
```

3. 核心实现方案

3.1 方案一：直接对接 Firecrawl API

3.1.1 Python 实现代码

```
代码块
1  import requests
2  import psycopg2
3  from sentence_transformers import SentenceTransformer
4  from bs4 import BeautifulSoup
5
6  # 配置参数
7  FIRE_CRAWL_API_KEY = "your_api_key"
8  OPENGAUSS_CONN_STR = "host=opengauss_host port=5432 dbname=vector_db user=user
   password=pass"
9
10 # 初始化模型
```

```python
model = SentenceTransformer('paraphrase-multilingual-MiniLM-L12-v2')

def crawl_and_store(url):
    # 调用Firecrawl API
    headers = {
        "Authorization": f"Bearer {FIRE_CRAWL_API_KEY}",
        "Content-Type": "application/json"
    }
    payload = {
        "url": url,
        "pageOptions": {
            "includeHtml": True
        }
    }

    response = requests.post(
        "https://api.firecrawl.dev/v0/scrape",
        headers=headers,
        json=payload
    )
    data = response.json()

    # 解析内容
    soup = BeautifulSoup(data['html'], 'html.parser')
    text_content = soup.get_text(separator=' ', strip=True)
    title = data.get('title', 'No Title')

    # 生成向量
    vector = model.encode(text_content)

    # 存储到openGauss
    conn = psycopg2.connect(OPENGAUSS_CONN_STR)
    cursor = conn.cursor()

    try:
        cursor.execute("""
            INSERT INTO crawled_data
            (url, title, content, vector, created_at)
            VALUES (%s, %s, %s, %s::vector, NOW())
            ON CONFLICT (url) DO UPDATE SET
            title = EXCLUDED.title,
            content = EXCLUDED.content,
            vector = EXCLUDED.vector
        """, (url, title, text_content, list(vector)))

        conn.commit()
        print(f"Successfully stored: {url}")
```

```
        except Exception as e:
            conn.rollback()
            print(f"Error storing {url}: {str(e)}")
        finally:
            cursor.close()
            conn.close()

# 示例使用
if __name__ == "__main__":
    crawl_and_store("https://example.com")
```

3.2 方案二：使用自托管 Firecrawl 服务

3.2.1 部署与集成

1. **部署 Firecrawl 服务**:

代码块

```
docker run -p 8000:8000 -e API_KEY=your_secret_key firecrawl/firecrawl
```

2. **修改对接代码**:

代码块

```
# 将API端点改为本地服务
FIRE_CRAWL_API = "http://localhost:8000"

response = requests.post(
    f"{FIRE_CRAWL_API}/v0/scrape",
    headers=headers,
    json=payload
)
```

4. 数据库准备

4.1 创建 openGauss 表结构

代码块

```
-- 启用向量扩展
CREATE EXTENSION vector;

-- 创建存储表
CREATE TABLE crawled_data (
```

```
    id SERIAL PRIMARY KEY,
    url VARCHAR(1024) UNIQUE,
    title TEXT,
    content TEXT,
    vector VECTOR(384),  -- 维度根据模型调整
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- 创建向量索引
CREATE INDEX ON crawled_data USING ivfflat (vector vector_l2_ops) WITH (lists =
 100);
```

5. 高级功能实现

5.1 批量爬取与存储

代码块

```
def batch_crawl(urls, batch_size=10):
    for i in range(0, len(urls), batch_size):
        batch = urls[i:i + batch_size]
        with ThreadPoolExecutor(max_workers=5) as executor:
            executor.map(crawl_and_store, batch)
```

5.2 增量爬取策略

代码块

```
def get_last_crawl_time(url):
    conn = psycopg2.connect(OPENGAUSS_CONN_STR)
    cursor = conn.cursor()
    cursor.execute("SELECT updated_at FROM crawled_data WHERE url = %s",
(url,))
    result = cursor.fetchone()
    cursor.close()
    conn.close()
    return result[0] if result else None

def smart_crawl(url):
    last_crawl = get_last_crawl_time(url)
    if last_crawl and (datetime.now() - last_crawl).days < 1:
        print(f"Skipping recently crawled URL: {url}")
        return

```

```
    crawl_and_store(url)
```

5.3 向量相似度搜索集成

代码块

```
def search_similar_content(query, top_k=5):
    conn = psycopg2.connect(OPENGAUSS_CONN_STR)
    cursor = conn.cursor()

    query_vector = model.encode(query)

    cursor.execute("""
        SELECT url, title, content,
               vector <-> %s::vector AS similarity
        FROM crawled_data
        ORDER BY similarity ASC
        LIMIT %s
    """, (list(query_vector), top_k))

    results = cursor.fetchall()
    cursor.close()
    conn.close()

    return [{
        'url': r[0],
        'title': r[1],
        'content': r[2][:200] + '...',  # 摘要
        'score': float(r[3])
    } for r in results]
```

6. 完整部署方案

6.1 Docker Compose 部署

代码块

```
version: '3.8'

services:
  firecrawl:
    image: firecrawl/firecrawl
    environment:
      API_KEY: your_firecrawl_key
    ports:
```

```yaml
      - "8000:8000"

  crawler:
    build: .
    environment:
      FIRE_CRAWL_API: "http://firecrawl:8000"
      OPENGAUSS_HOST: "opengauss"
    depends_on:
      - firecrawl
      - opengauss

  opengauss:
    image: opengauss:3.0
    environment:
      GS_PASSWORD: "OpenGauss@123"
    ports:
      - "5432:5432"
    volumes:
      - opengauss_data:/var/lib/opengauss

volumes:
  opengauss_data:
```

6.2 自动化爬取调度

使用 Celery 实现定时任务：

代码块

```python
from celery import Celery

app = Celery('crawler', broker='redis://redis:6379/0')

@app.task
def scheduled_crawl(url):
    crawl_and_store(url)

# 配置定时任务
app.conf.beat_schedule = {
    'daily-crawl': {
        'task': 'scheduled_crawl',
        'schedule': 86400.0,  # 每天
        'args': ('https://example.com',)
    },
}
```

四、VectorETL 对接 openGauss 实现方案

1. 整体架构设计

VectorETL 与 openGauss 的集成主要通过以下数据处理流程实现：

```
[原始数据源] → [VectorETL 处理] → [向量化转换] → [openGauss 向量存储] → [向量检索服务]
```

2. 前置准备

2.1 环境要求

- VectorETL 运行环境（Python 3.8+ 或 Docker）
- openGauss 3.0+ 数据库（已安装 vector 扩展）
- 向量模型服务（如 Sentence-Transformers、HuggingFace 或自定义模型）
- 网络互通性验证

2.2 依赖安装

```
# Python 环境
pip install vector-etl opengauss-python psycopg2-binary numpy sentence-transformers pandas

# 或使用 Docker
docker pull vectordotdev/vector-etl
```

3. 核心实现方案

3.1 方案一：使用 Python SDK 直接对接

3.1.1 基础实现代码

```
from vector_etl import VectorETL
import psycopg2
from sentence_transformers import SentenceTransformer
import numpy as np
```

```python

# 配置参数
OPENGAUSS_CONN_STR = "host=opengauss_host port=5432 dbname=vector_db user=user password=pass"
MODEL_NAME = "paraphrase-multilingual-MiniLM-L12-v2"

# 初始化组件
model = SentenceTransformer(MODEL_NAME)
etl = VectorETL()

def process_and_store(data_source):
    # 1. 数据抽取
    raw_data = etl.extract(data_source)

    # 2. 数据转换
    processed_data = []
    for item in raw_data:
        # 文本清洗和预处理
        clean_text = etl.clean_text(item['content'])

        # 向量化
        vector = model.encode(clean_text)

        processed_data.append({
            'id': item['id'],
            'content': clean_text,
            'metadata': item.get('metadata', {}),
            'vector': vector
        })

    # 3. 加载到openGauss
    conn = psycopg2.connect(OPENGAUSS_CONN_STR)
    cursor = conn.cursor()

    try:
        # 批量插入
        records = [(d['id'], d['content'], d['metadata'], list(d['vector'])) for d in processed_data]
        cursor.executemany("""
            INSERT INTO vector_docs (id, content, metadata, vector)
            VALUES (%s, %s, %s::jsonb, %s::vector)
            ON CONFLICT (id) DO UPDATE SET
            content = EXCLUDED.content,
            metadata = EXCLUDED.metadata,
            vector = EXCLUDED.vector
        """, records)

```

```
            conn.commit()
            print(f"成功插入/更新 {len(processed_data)} 条记录")
        except Exception as e:
            conn.rollback()
            print(f"数据库操作失败: {str(e)}")
        finally:
            cursor.close()
            conn.close()

# 示例使用
if __name__ == "__main__":
    data_source = {
        "type": "csv",
        "path": "data/documents.csv",
        "options": {"delimiter": ","}
    }
    process_and_store(data_source)
```

3.2 方案二：使用 VectorETL 配置文件

3.2.1 配置 YAML 文件（`vector-etl-config.yaml`）

代码块

```
version: "1"
sources:
  - name: "document_source"
    type: "csv"
    path: "data/documents.csv"
    options:
      delimiter: ","

transformations:
  - name: "text_clean"
    type: "text_processing"
    operations:
      - remove_special_chars
      - normalize_whitespace
      - lowercase

  - name: "vectorize"
    type: "embedding"
    model: "sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2"
    input_field: "content"
    output_field: "vector"

```

```yaml
sinks:
  - name: "opengauss_sink"
    type: "opengauss"
    connection:
      host: "opengauss_host"
      port: 5432
      database: "vector_db"
      user: "user"
      password: "pass"
    table: "vector_docs"
    mapping:
      id: "id"
      content: "content"
      metadata: "metadata"
      vector: "vector"
    batch_size: 1000
```

3.2.2 运行命令

代码块

```
vector-etl --config vector-etl-config.yaml
```

4. 数据库准备

4.1 创建 openGauss 表结构

代码块

```
-- 启用向量扩展
CREATE EXTENSION vector;

-- 创建存储表
CREATE TABLE vector_docs (
    id VARCHAR(128) PRIMARY KEY,
    content TEXT,
    metadata JSONB,
    vector VECTOR(384),  -- 维度根据模型调整
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- 创建向量索引
CREATE INDEX ON vector_docs USING ivfflat (vector vector_l2_ops) WITH (lists =
100);
```

```
16
17  -- 创建全文检索索引（可选）
18  CREATE EXTENSION pg_trgm;
19  CREATE INDEX ON vector_docs USING gin (content gin_trgm_ops);
```

5. 高级功能实现

5.1 增量数据处理

代码块

```
1   def get_last_processed_id():
2       conn = psycopg2.connect(OPENGAUSS_CONN_STR)
3       cursor = conn.cursor()
4       cursor.execute("SELECT MAX(id) FROM vector_docs")
5       last_id = cursor.fetchone()[0]
6       cursor.close()
7       conn.close()
8       return last_id
9
10  def incremental_update(data_source):
11      last_id = get_last_processed_id()
12
13      # 修改VectorETL提取逻辑，只获取新数据
14      raw_data = etl.extract(
15          data_source,
16          filter_logic=f"id > '{last_id}'" if last_id else None
17      )
18
19      if raw_data:
20          process_and_store(raw_data)
```

5.2 多模型向量支持

代码块

```
1   class MultiVectorETL:
2       def __init__(self):
3           self.text_model = SentenceTransformer('paraphrase-MiniLM-L6-v2')
4           self.image_model = load_image_model()  # 自定义图像模型
5
6       def process_item(self, item):
7           vectors = {}
8
9           if item['type'] == 'text':
```

```
            vectors['text_vector'] = self.text_model.encode(item['content'])
        elif item['type'] == 'image':
            vectors['image_vector'] = self.image_model.encode(item['image_data'])

        return {**item, **vectors}

# 在存储时需要调整表结构和SQL
```

5.3 分布式处理

代码块

```
from multiprocessing import Pool

def parallel_process(data_chunk):
    # 每个进程独立的数据库连接
    local_etl = VectorETL()
    local_model = SentenceTransformer(MODEL_NAME)
    # 处理逻辑...
    return processed_chunk

if __name__ == "__main__":
    raw_data = etl.extract(data_source)
    chunks = np.array_split(raw_data, 4)  # 分为4份

    with Pool(processes=4) as pool:
        results = pool.map(parallel_process, chunks)

    # 合并结果并存储
    all_processed = [item for chunk in results for item in chunk]
    batch_insert(all_processed)
```

6. 完整部署方案

6.1 Docker Compose 部署

代码块

```
version: '3.8'

services:
  vector-etl:
    image: vectordotdev/vector-etl:latest
    volumes:
```

```
      - ./config:/app/config
      - ./data:/app/data
    environment:
      CONFIG_PATH: "/app/config/vector-etl-config.yaml"
    depends_on:
      - opengauss

  opengauss:
    image: opengauss:3.0
    environment:
      GS_PASSWORD: "OpenGauss@123"
    ports:
      - "5432:5432"
    volumes:
      - opengauss_data:/var/lib/opengauss

volumes:
  opengauss_data:
```

6.2 Kubernetes 部署示例

代码块

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: vector-etl
spec:
  replicas: 3
  selector:
    matchLabels:
      app: vector-etl
  template:
    metadata:
      labels:
        app: vector-etl
    spec:
      containers:
      - name: vector-etl
        image: vectordotdev/vector-etl:latest
        volumeMounts:
        - name: config
          mountPath: /app/config
        env:
        - name: CONFIG_PATH
          value: "/app/config/vector-etl-config.yaml"
```

```
        volumes:
        - name: config
          configMap:
            name: vector-etl-config
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: vector-etl-config
data:
  vector-etl-config.yaml: |
    # 这里放置之前的YAML配置内容
```

五、项目开发时间规划

1. 项目总体时间安排

总周期：2025年7月1日 - 2025年9月30日（共13周）

2. 详细时间计划表

第一阶段：环境准备与基础部署 (2025.07.01-2025.07.14)

任务	时间	交付物
1.1 openGauss Docker化部署研究	07.01-07.03	部署方案文档
1.2 Docker环境配置与测试	07.04-07.07	可运行的Docker环境
1.3 openGauss向量扩展验证	07.08-07.10	测试报告
1.4 Kafka集群搭建	07.11-07.12	Kafka测试环境
1.5 Firecrawl环境准备	07.13-07.14	爬虫测试环境

阶段目标：完成所有基础环境搭建并通过验证

第二阶段：核心功能开发 (2025.07.15-2025.08.25)

模块一：Kafka集成 (07.15-07.28)

任务	时间	交付物

2.1.1 Kafka Connect配置开发	07.15-07.18	连接器配置文件
2.1.2 自定义消费者实现	07.19-07.22	Java消费者代码
2.1.3 向量数据特殊处理	07.23-07.25	向量转换模块
2.1.4 性能测试与优化	07.26-07.28	性能测试报告

模块二：Firecrawl集成 (07.29-08.11)

任务	时间	交付物
2.2.1 网页抓取管道开发	07.29-08.01	Python爬取脚本
2.2.2 内容向量化处理	08.02-08.05	向量化服务
2.2.3 增量爬取机制	08.06-08.08	增量处理模块
2.2.4 分布式爬取测试	08.09-08.11	分布式方案文档

模块三：VectorETL集成 (08.12-08.25)

任务	时间	交付物
2.3.1 ETL流程设计	08.12-08.14	ETL流程图
2.3.2 多模型向量支持	08.15-08.18	多模型适配器
2.3.3 批量处理优化	08.19-08.22	批量处理模块
2.3.4 端到端测试	08.23-08.25	集成测试报告

阶段目标：完成所有核心功能模块开发并通过集成测试

第三阶段：文档与优化 (2025.08.26-2025.09.15)

任务	时间	交付物
3.1 部署指南编写	08.26-08.28	部署文档
3.2 最佳实践手册	08.29-09.01	实践指南
3.3 性能调优白皮书	09.02-09.05	优化方案

3.4 常见问题整理	09.06-09.08	FAQ文档
3.5 安全配置指南	09.09-09.11	安全手册
3.6 代码审查与重构	09.12-09.15	优化后的代码

阶段目标：完成所有技术文档并优化代码质量

第四阶段：社区提交与收尾 (2025.09.16-2025.09.30)

任务	时间	交付物
4.1 文档格式转换	09.16-09.18	社区标准文档
4.2 示例代码整理	09.19-09.21	可运行示例包
4.3 社区PR提交	09.22-09.24	提交记录
4.4 根据反馈修改	09.25-09.27	修订版本
4.5 项目总结报告	09.28-09.30	总结报告

阶段目标：完成社区贡献并通过验收

3. 关键里程碑节点

1. **07.14** - 基础环境验收
2. **07.28** - Kafka集成完成
3. **08.11** - Firecrawl集成完成
4. **08.25** - VectorETL集成完成
5. **09.15** - 文档初稿完成
6. **09.30** - 社区提交完成

4. 每周工作时间安排

- **工作日**：每天6小时（9:00-12:00，14:00-17:00）
- **总开发时间**：约400小时（13周×30小时）

5. 风险管理计划

风险	应对措施	负责人

组件兼容性问题	提前建立版本矩阵	本人
性能不达标	预留2周优化时间	本人
文档不符合要求	提前与社区沟通格式	本人
个人时间冲突	关键阶段增加周末工时	本人

6. 开发环境配置

- **硬件**：笔记本(i7/16GB/512GB SSD) + 云测试服务器(4C8G)
- **软件**：
  - Docker Desktop
  - IntelliJ IDEA/PyCharm
  - PostgreSQL/OpenGauss客户端工具
  - Kafka/Zookeeper套件

7. 质量保证措施

1. 每周日进行代码审查
2. 每个模块完成后进行单元测试
3. 关键功能进行性能压测
4. 文档经过三次校审