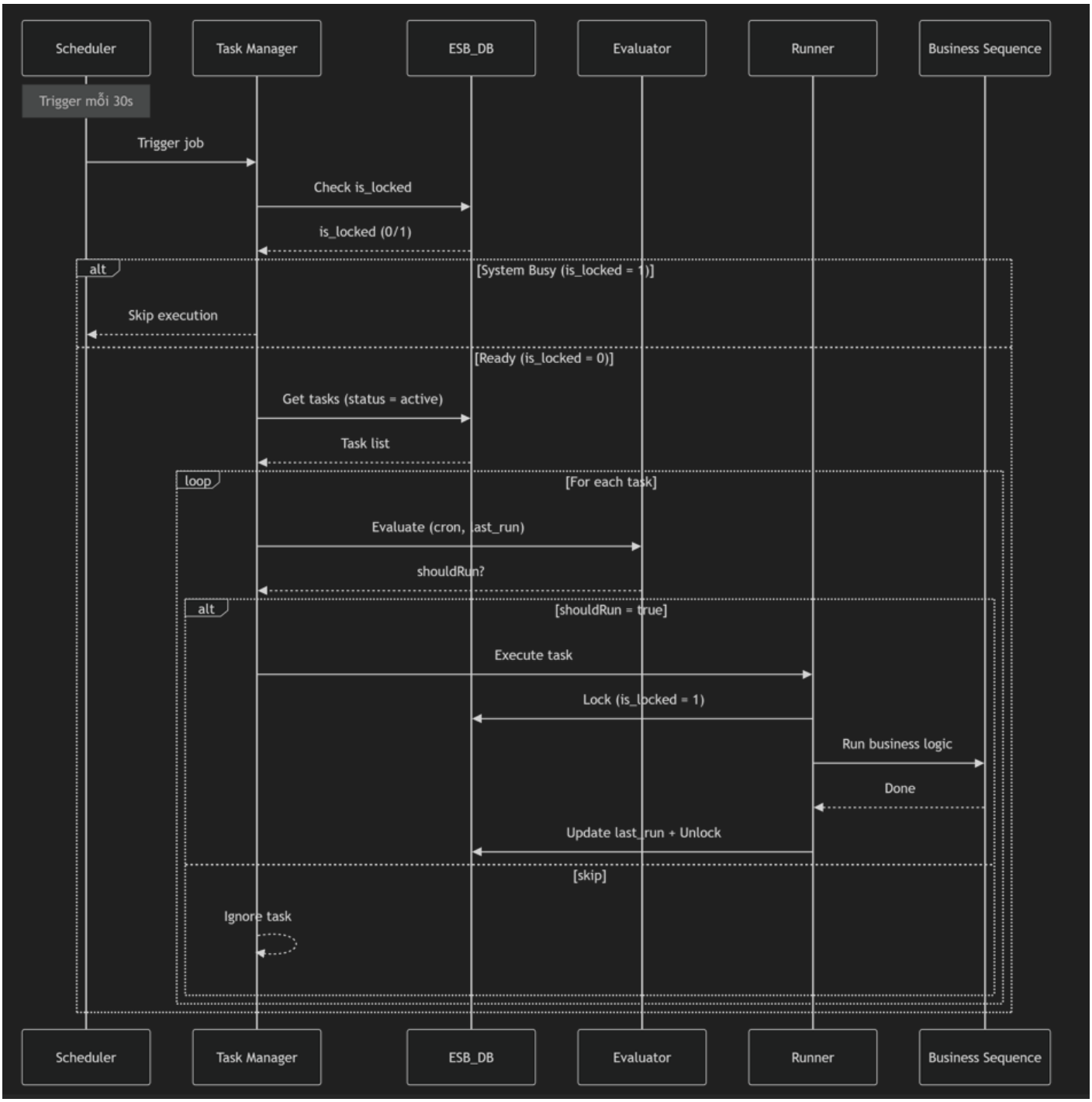
# CHƯƠNG III: CƠ CHẾ ĐỒNG BỘ DỮ LIỆU TỰ ĐỘNG

Xử lý đồng bộ dữ liệu lớn

# TRANG 1: Sơ đồ luồng hoạt động tổng quát (Sequence Diagram)

Hệ thống đồng bộ của LGSP không chạy theo lịch cứng mà chạy theo cơ chế "Quét và Đánh giá" (Scan & Evaluate) dựa trên cấu hình linh hoạt trong Database.




Sau khi Task kích hoạt, hệ thống sẽ đi qua một chuỗi các Sequence có nhiệm vụ chuyên biệt để đảm bảo tính an toàn và linh hoạt.

# 1.1. Bước 1: Quản lý Luồng (ManagerEvaluateTasksSequence)

Đây là sequence "cửa ngõ". Nhiệm vụ chính:

- **Kiểm soát concurrency:** Sử dụng `dataServiceCall` tới bảng `sync_lock`. Nếu có một tiến trình `RunTaskSequence` trước đó chưa kết thúc (vẫn đang `is_locked=1`), nó sẽ dừng ngay lập tức.
- **Phân tách dữ liệu:** Sử dụng mediator `<iterate>` để biến danh sách Task từ Database thành từng gói tin độc lập cho mỗi Task.

# 1.2. Bước 2: Đánh giá Cron (EvaluateSingleTaskSequence)

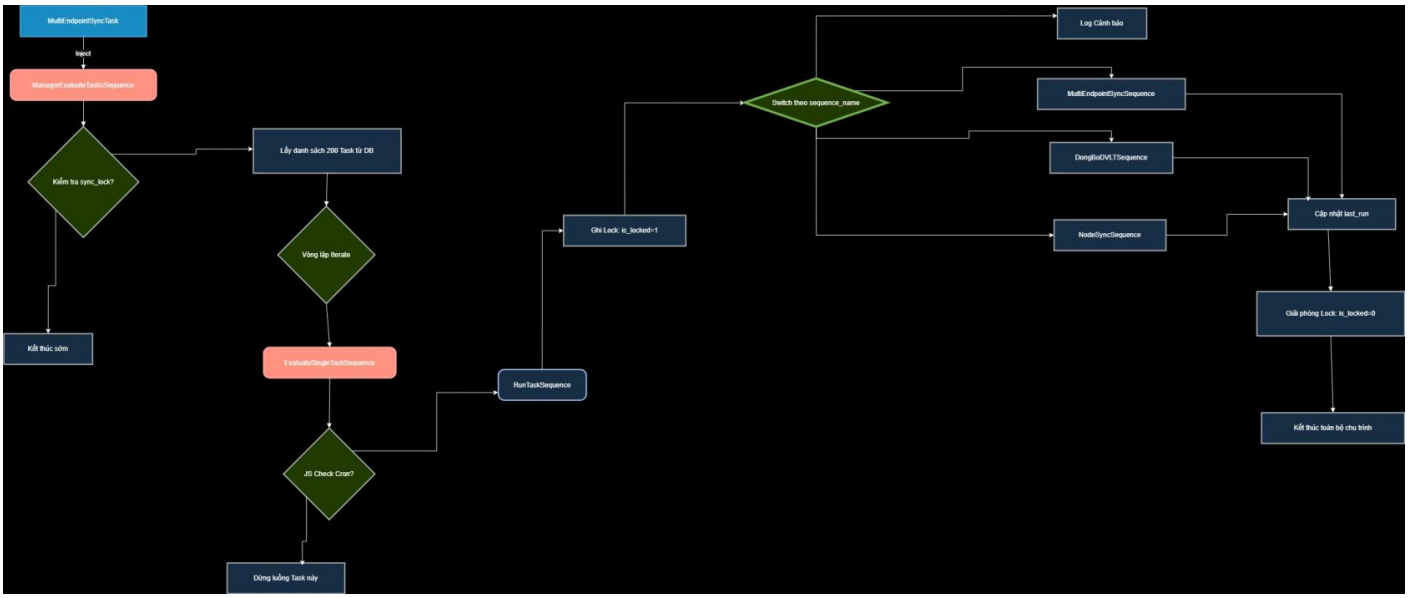
Với mỗi Task được tách ra, bộ Evaluator sẽ:

- Trích xuất `cron_expression` (ví dụ: `0 */15 * * * ?` - chạy mỗi 15 phút).
- Sử dụng **Script Mediator (JavaScript)** để tính toán thời gian chạy tiếp theo.
- **Kết quả:** Nếu thỏa mãn điều kiện thời gian, nó sẽ nạp biến `shouldRun = true` vào Message Context.

# 1.3. Bước 3: Bộ định tuyến động (RunTaskSequence)

Sequence này đóng vai trò là "Cầu giao điện" tổng. Nó thực hiện:

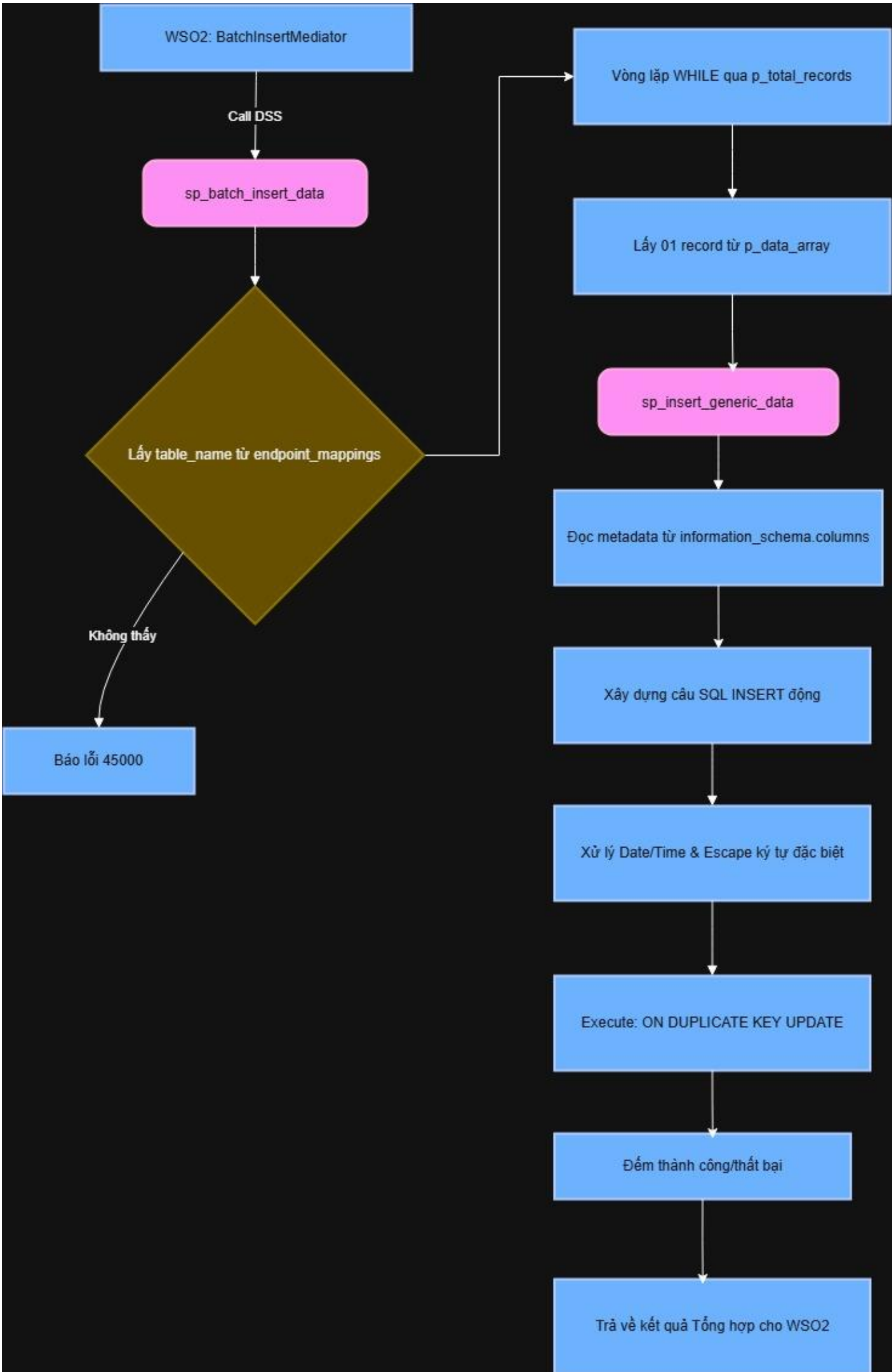
- **Khóa hệ thống:** Gọi `lockSync` để ngăn các Task khác nhảy vào chiếm dụng tài nguyên.
- **Điều hướng (Switch Mediator):** Dựa vào tên Sequence lưu trong database ( `sequence_name`), nó sẽ điều hướng tới logic nghiệp vụ thật sự:
  - `MultiEndpointSyncSequence`: Đồng bộ hóa đa điểm tiêu chuẩn.
  - `DongBoDVLTSequence`: Đồng bộ dữ liệu dịch vụ lưu trú (Sử dụng `BatchInsertMediator` để nạp dữ liệu lớn).
  - `DongboVBDenEdocSequence`: Chuyên trách đồng bộ Văn bản điện tử.
- **Hoàn tất:** Sau khi nghiệp vụ chạy xong, nó cập nhật `last_run` vào DB và thực hiện `unlockSync`.




# 1.4. Cơ chế nạp dữ liệu động (Dynamic Batch Insert)

Trong dự án LGSP, thay vì viết hàng trăm câu lệnh `INSERT` cho hàng trăm loại danh mục khác nhau, chúng ta sử dụng cơ chế **Dynamic SQL** thông qua 2 Stored Procedure phối hợp. Cơ chế này cho phép hệ thống tự động nhận diện cấu trúc tệp JSON trả về từ đối tác và nạp vào bảng tương ứng trong Database.

# 1.4.1. Sơ đồ luồng xử lý tại Database


WSO2: BatchInsertMediator
Call DSS
sp_batch_insert_data
Lấy table_name từ endpoint_mappings
Không thấy
Báo lỗi 45000
Vòng lặp WHILE qua p_total_records
Lấy 01 record từ p_data_array
sp_insert_generic_data
Đọc metadata từ information_schema.columns
Xây dựng câu SQL INSERT động
Xử lý Date/Time & Escape ký tự đặc biệt
Execute: ON DUPLICATE KEY UPDATE
Đếm thành công/thất bại
Trả về kết quả Tổng hợp cho WSO2

# 1.4.2. Phân tích Procedure bộ điều phối: `sp_batch_insert_data`

Nhiệm vụ chính: **Phân rã mảng dữ liệu (Array Parsing)**.

- **Tham số đầu vào:** `p_endpoint_path` (để xác định bảng đích), `p_data_array` (chuỗi JSON khổng lồ chứa hàng trăm bản ghi), `p_total_records`.
- **Cơ chế chịu lỗi (Fault Tolerance):** Procedure sử dụng `DECLARE CONTINUE HANDLER FOR SQLEXCEPTION`. Điều này cực kỳ quan trọng: Nếu trong 100 bản ghi có 1 bản ghi bị lỗi dữ liệu, Procedure sẽ **không dừng lại** mà chỉ tăng biến `v_error_count`, sau đó tiếp tục xử lý bản ghi tiếp theo.
- **Kết quả trả về:** Một tập bản ghi (ResultSet) chứa: `status` (success, partial, error), `message` chi tiết và `record_count` thực tế đã nạp thành công.

# 1.4.3. Phân tích Procedure thực thi động: `sp_insert_generic_data`

Đây là chìa khóa của sự linh hoạt. Nó thực hiện **Tự động nhận diện cấu trúc bảng (Table Discovery)**.

- **Bước bóc tách dữ liệu:** Procedure kiểm tra xem JSON có bọc trong thẻ `$.data` hay không để lấy đúng nội dung cần thiết.
- **Bước ánh xạ cột (Dynamic Mapping):**
  - Nó truy cập vào `information_schema.columns` để xem bảng đích có những cột nào, kiểu dữ liệu là gì.
  - Nếu cột trong JSON không có trong Database -> Tự động bỏ qua (Giúp hệ thống không bị chết khi đối tác thêm trường dữ liệu mới).
- **Xử lý kiểu dữ liệu:**
  - Tự động nhận diện các cột `DATE`, `DATETIME`, `TIMESTAMP` để sử dụng hàm `STR_TO_DATE`.
  - Tự động thay thế (`REPLACE`) các ký tự đặc biệt như dấu nháy kép, dấu gạch chéo để chống lỗi cú pháp SQL.
- **Cơ chế Upsert:** Sử dụng cú pháp `ON DUPLICATE KEY UPDATE updated_at = NOW()`.
  - Nếu bản ghi đã tồn tại (trùng Primary Key): Cập nhật thông tin mới nhất.
  - Nếu chưa có: Chèn mới.
  - Điều này giúp dữ liệu trong kho địa phương luôn đồng bộ với trục Quốc gia mà không gây trùng lặp.

# 1.4.4. Các bảng cấu hình phụ thuộc

Để 2 Procedure này hoạt động, bạn cần cấu hình bảng mapping

```
CREATE TABLE endpoint_mappings (
    id INT AUTO_INCREMENT PRIMARY KEY,
    endpoint_path VARCHAR(200) NOT NULL, -- Đường dẫn API (Vd: /api/v1/don-vi)
    table_name VARCHAR(100) NOT NULL,    -- Tên bảng trong Database
    status TINYINT DEFAULT 1             -- 1: Kích hoạt
);
```

# TRANG 2: Bộ máy điều phối Task (Orchestrator)

## 2.1. MultiEndpointSyncTask (Điểm khởi đầu)

Nằm tại: `src/main/wso2mi/artifacts/tasks/MultiEndpointSyncTask.xml`.

- **Class:** `org.apache.synapse.startup.tasks.MessageInjector`.
- **Cơ chế:** Task này không chứa logic đồng bộ. Nó đóng vai trò "còi báo động" định kỳ cứ mỗi 30 giây (hoặc cấu hình) sẽ đẩy một thông điệp vào bộ máy xử lý.

## 2.2. ManagerEvaluateTasksSequence (Bộ quản lý)

Đây là "bộ não" điều phối toàn bộ quá trình:

1. **Kiểm tra Khóa (sync_lock):** Hệ thống gọi `syncLockDataService/getLock`. Nếu một luồng đồng bộ khác đang chạy, luồng mới sẽ tự động hủy để tránh tình trạng "Race Condition" (nhiều tiến trình cùng ghi vào một bảng dữ liệu).
2. **Lấy cấu hình Task:** Truy vấn lên tới 200 Task đang ở trạng thái kích hoạt (`status='1'`) từ bảng `job_schedule`.
3. **Lặp (Iterate):** Sử dụng thẻ `<iterate>` của WSO2 để chia nhỏ danh sách Task cho các luồng xử lý song song, tối ưu hóa hiệu suất CPU.

# TRANG 3: Phân tích Logic Đánh giá Cron Expression (JS Engine)

## 3.1. Tại sao cần JavaScript?

WSO2 MI mặc định không có hàm so sánh thời gian với biểu thức Cron phức tạp (như `0 0/15 * * * ?`). Do đó, dự án sử dụng **JavaScript (Nashorn Engine)** để gọi trực tiếp các thư viện Java của Quarkz.

## 3.2. Chi tiết mã nguồn Evaluator

Tại `EvaluateSingleTaskSequence.xml`, hệ thống thực hiện các bước:

1. Lấy `cron_expression` và `last_run` (chuỗi string) từ Database.
2. Chuyển đổi `last_run` sang đối tượng `Date` của Java.
3. Sử dụng class `org.quartz.CronExpression` để tính toán thời điểm chạy tiếp theo (`nextRun`).
4. **So sánh:** Nếu `nextRun` nhỏ hơn hoặc bằng thời gian hiện tại (`now`), hệ thống xác định Task này đã đến hạn chạy (`shouldRun = true`).

# TRANG 4: Thực thi và Cập nhật trạng thái (Runner)

## 4.1. RunTaskSequence (Bộ thực thi)

Ngay khi nhận được tín hiệu `shouldRun = true`, Runner sẽ thực hiện:

- **Locking:** Gọi `lockSync` để đánh dấu hệ thống đang bận thực hiện nghiệp vụ nặng.
- **Dynamic Call:** Sử dụng `sequence_name` lấy từ DB để gọi đến Sequence xử lý nghiệp vụ thật sự (Vd: `MultiEndpointSyncSequence`). Điều này cho phép mở rộng hệ thống bằng cách chỉ cần thêm Sequence mới và cấu hình vào DB mà không cần sửa code Runner.

## 4.2. Hoàn tất và Giải phóng

Sau khi nghiệp vụ đồng bộ hoàn thành (thành công hoặc thất bại):

- **Cập nhật Last Run:** Lưu timestamp chạy thành công cuối cùng vào DB để lượt quét sau tính toán đúng.
- **Unlock:** Giải phóng khóa (`is_locked = 0`) để các đợt quét định kỳ tiếp theo có thể thực hiện các Task khác.