# MuleSoft Design Patterns

## Integration Architecture  Advanced Flows


*Scalability · Reliability · Reusability*

### Topics Covered:

API-Led Connectivity · Batch Processing
Scatter-Gather · System Sync · Large Files

*Prepared for:*
**Senior MuleSoft Developer / Architect**
December 9, 2025

# Contents

# Chapter 1

# API-Led Connectivity Pattern

## 1.1 Definition & Purpose

API-Led connectivity is an architectural approach that shifts away from point-to-point integration to a structured, reusable, three-layered architecture.

**Purpose:**

- **Decoupling:** Changes in the backend (e.g., swapping SAP for Oracle) do not break the frontend apps.
- **Reusability:** A "Customer API" can be reused by Mobile, Web, and B2B systems.
- **Agility:** Different teams can work on different layers simultaneously.

## 1.2 Architecture Diagram

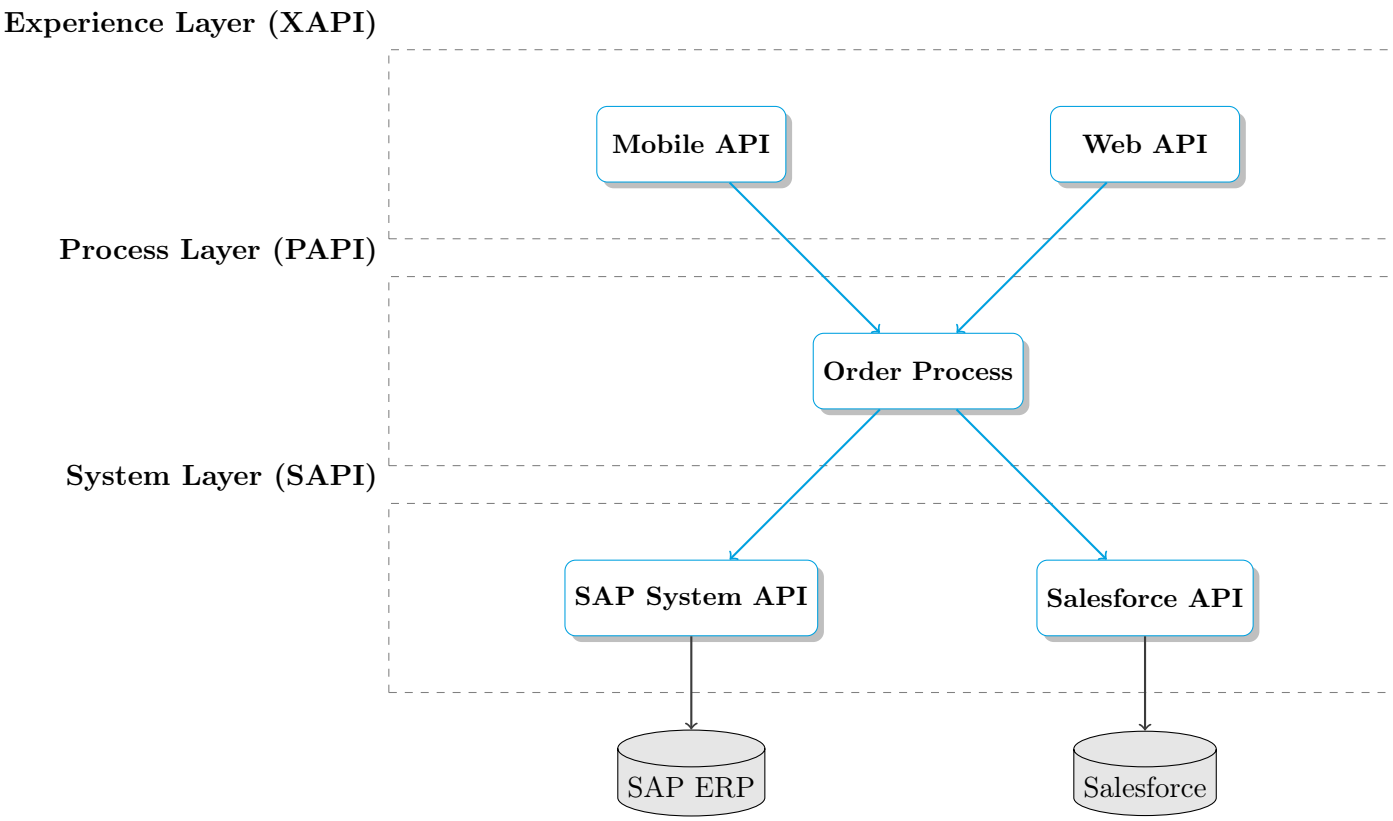



Figure 1.1: The Three Layers of API-Led Connectivity

## 1.3 Best Practices

- **XAPI:** Should contain NO business logic. Only format conversion (JSON/XML).
- **PAPI:** Where the logic lives (aggregation, orchestration, routing).
- **SAPI:** Should mirror the backend system. Simple CRUD operations.

# Chapter 2

# Scatter-Gather (Parallel Processing)

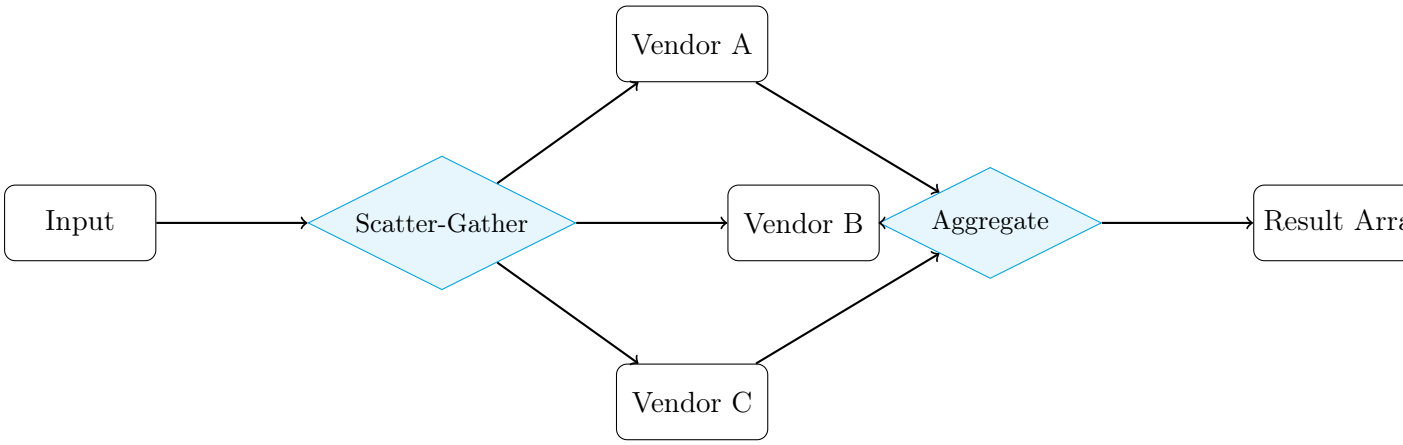
## 2.1 Definition

The **Scatter-Gather** router sends a message to multiple routes *concurrently* (in parallel) and waits for all of them to complete. It then aggregates the results into a single payload.

## 2.2 Internal Working

1. **Multicasting:** The event is cloned (deep copy) and sent to each route.
2. **Threading:** Each route runs in a separate thread (CPU-Light or I/O-Blocking pools).
3. **Aggregation:** The output is a Mule Message Object containing a collection of results.

## 2.3 Architecture Diagram




## 2.4 Handling the Output (DataWeave)

The output of Scatter-Gather is an Object where keys are numeric indices (0, 1, 2...). We often need to flatten this.

```
%dw 2.0
output application/json
---
// This extracts just the payloads from the multipart structure
flatten(payload..payload)
```

Listing 2.1: Processing Scatter-Gather Output

## 2.5 Common Mistakes

**Failure Behavior**

If **one** route fails, the entire Scatter-Gather fails and throws a `MULE:COMPOSITE_ROUTING` error.

**Solution:** Wrap *each* route inside a **Try Scope** with an `On Error Continue`. This ensures that if Vendor A fails, Vendor B and C still return data, and you handle the partial failure in the aggregation step.

# Chapter 3

# Batch Processing (ETL)

## 3.1 When to use it?

Use Batch Processing when dealing with:

- Large datasets (e.g., 100,000+ records).
- Data synchronization (ETL).
- Reliability is key (handling individual record failures).

## 3.2 Internal Architecture (Phases)

1. **Load Dispatch:** The payload is split into individual records and stored in a persistent queue (serialization).
2. **Process (Steps):** Records are processed asynchronously in blocks (default block size: 100).
3. **On Complete:** A summary report is generated (Total records, Success count, Failure count).

## 3.3 Mule XML Configuration

```
<batch:job name="syncContactsBatch" maxFailedRecords="-1">

    <batch:process-records>
        <batch:step name="ValidateStep">
            <ee:transform>
                <ee:message>
                    <ee:set-payload><![CDATA[%dw 2.0
                    output application/java
                    ---
                    payload update { case .email -> lower(.) }
                    ]]></ee:set-payload>
                </ee:message>
            </ee:transform>
        </batch:step>

        <batch:step name="InsertStep" accept-policy="ONLY_FAILURES">
             </batch:step>

        <batch:step name="BulkInsertStep">
```

```
            <batch:aggregator doc:name="Batch Aggregator" size="100">
                <db:insert doc:name="Bulk Insert" ... />
            </batch:aggregator>
        </batch:step>
    </batch:process-records>

    <batch:on-complete>
        <logger message="Batch Finished: #[payload]"/>
    </batch:on-complete>

</batch:job>
```

Listing 3.1: Batch Job Structure

**Batch Aggregator**

Always use **Batch Aggregator** for Database or Salesforce inserts. Inserting 100 records in 1 API call is significantly faster than 100 separate API calls.

# Chapter 4

# System Synchronization & Watermarking

## 4.1 The Concept

Synchronizing data between two systems (e.g., pulling new employees from HR to Active Directory) requires keeping track of what has already been processed.

## 4.2 Watermarking (Object Store)

Watermarking is the technique of storing the timestamp or ID of the last processed record.

### 4.2.1 Flow Logic

1. **Retrieve:** Get ‘lastModifiedDate‘ from Object Store. (Default to 1900-01-01 if null).
2. **Query:** SELECT * FROM Users WHERE ModifiedDate > ‘vars.watermark‘.
3. **Process:** Sync the records.
4. **Store:** Update the Object Store with the ‘max(ModifiedDate)‘ from the current batch.

## 4.3 Scheduler

Synchronization is usually triggered by a **Scheduler** component (e.g., run every 15 minutes).

**Fixed Frequency vs Cron**

- **Fixed Frequency:** Runs every X minutes from the *end* of the last run.
- **Cron:** Runs at specific wall-clock times (e.g., every Friday at 5 PM). Beware of overlapping runs if the process takes too long!

# Chapter 5

# Large File Processing

## 5.1 The Challenge: Memory

Processing a 2GB CSV file in a standard JVM (which might only have 1GB heap) will cause an 'OutOfMemoryError'.

## 5.2 Streaming Strategies

Mule 4 uses streaming by default, but for large files, specific configurations ensure stability.

### 5.2.1 1. Repeatable File Store Stream

Mule keeps a small buffer in memory and writes the rest to a temporary file on disk. This allows you to read a stream multiple times without crashing RAM.

```
<file:config name="File_Config">
    <file:connection workingDir="/tmp/input" />
</file:config>

<ee:transform>
    <ee:message>
        <ee:set-payload><![CDATA[%dw 2.0
        output application/json deferred=true
        ---
        payload map (item) -> { ... }
        ]]></ee:set-payload>
    </ee:message>
</ee:transform>
```

Listing 5.1: Streaming Config

## 5.3 Using For-Each vs. Batch

For large files (CSV/XML):

- **Batch:** Best for record-level error handling and ETL. Overhead of serialization.
- **For-Each:** Lighter. Use with `batchSize` (e.g., split 1000 records at a time) to avoid loading everything.

# Chapter 6

# Interview Guide

## 6.1 Critical Interview Questions

### 6.1.1 Q1: How does Scatter-Gather handle variables?

**Answer:** Variables set *inside* a route are **local** to that route. They are NOT propagated to the aggregation phase or other routes. Only the payload and attributes are aggregated.

### 6.1.2 Q2: What is the difference between For-Each and Batch?

**Answer:**

- **For-Each:** Synchronous, single-threaded. If one fails, the loop stops (unless Try-Catch used). No persistent queuing.
- **Batch:** Asynchronous, multi-threaded. Persistent queues. Handles failures gracefully (continues to next record). Includes reporting phase.

### 6.1.3 Q3: How do you handle a "Composite Routing Error" in Scatter-Gather?

**Answer:** This error occurs when one route fails. To access the partial success data, checking the error object is required, but the best practice is to wrap every route in a Try Scope with On-Error-Continue, returning a specific structure (e.g., 'success: false, error: ...') so the aggregator can filter them out manually.

## 6.2 Mini Project: Sync Service

**Scenario:** Sync Contacts from Database to Salesforce every 10 mins.

1. **Source:** Scheduler (10 min).
2. **Logic:** Retrieve Watermark -> Select DB (WHERE date > watermark).
3. **Processing:** Use **Batch Job**.
4. **Batch Step 1:** Transform DB structure to Salesforce JSON.
5. **Batch Step 2 (Aggregator):** Use Salesforce Create-Bulk connector (Commit size 200).
6. **On Complete:** Update Watermark in Object Store.