Chapter 3: API & Log Data Ingestion (Week 3)

Introduction: Broadening Our Data Horizons

The Paradigm Shift from OLTP to a Multi-Source World

In the previous week, the focus was on the foundational bedrock of data systems: relational databases. The skills acquired—the ability to query, connect to, and extract data using SQL and Python—are indispensable for any aspiring data professional.[1] This paradigm, which centers on Online Transaction Processing (OLTP) systems, is a critical component of data architecture. However, in today's digital economy, relying solely on structured, relational data would be a significant oversight. The modern data landscape is a dynamic, complex tapestry woven from a multitude of sources, including vast amounts of semi-structured and unstructured data.

This lecture marks a crucial paradigm shift in the understanding of data sources. It is no longer sufficient to treat data as a collection of neatly organized tables. Instead, data must be conceptualized as an expansive ecosystem, one that includes the free-flowing streams of information from web APIs and the detailed, event-driven records stored in log files. The ability to effectively navigate and ingest data from these new frontiers is what separates a traditional data analyst from a modern data engineer. The core objective is to expand the repertoire of data sources beyond the confines of relational databases, embracing a world where data resides on web servers, in cloud services, and within application logs.[1]

The Role of the Data Engineer in a Converging Data Ecosystem

The traditional role of the data engineer was often defined by the Extract, Transform, Load (ETL) process, where data was extracted from a source, meticulously transformed, and then loaded into a target data warehouse for analysis. While this remains a vital skill, the demands of a multi-source environment have evolved this role. The modern data engineer is a versatile architect who designs and manages sophisticated data pipelines that can ingest information from a myriad of origins, including real-time data streams from sensors and application logs.[2]

This course, DSI310: Data Exploration and Preprocessing, is designed to prepare students for this new reality. The focus on a multi-source data pipeline is central to the curriculum, demonstrating that all data—whether from a legacy database, a web API, or a log file—must be able to flow into a unified repository.[1] The ultimate goal is to create a robust, scalable data platform that can handle this convergence, transforming raw, disparate data into structured, actionable insights for business intelligence and machine learning applications.[2]

Section 1: The First Pillar of Modern Ingestion - Web APIs

1.1 What is an API? A Conceptual Deep Dive

At its essence, an Application Programming Interface (API) is a messenger that takes requests and tells a system what to do, then returns the system's response back to the user. Think of an API as a waiter in a restaurant. A customer (your Python program) looks at a menu (the API's documentation) and tells the waiter (the API) what they want (a request). The waiter goes to the kitchen (the server or application) to get the food (the data) and brings it back to the customer. This abstraction allows applications to communicate and share data without needing to understand the underlying complexity of each other's code.

While many types of APIs exist, the most prevalent for web data exchange is a RESTful API (Representational State Transfer). REST APIs use standard HTTP methods like GET, POST, PUT, and DELETE to perform operations on resources, which are typically identified by unique URLs. A GET request, for example, is used to retrieve data, such as a list of customer records from a /customers endpoint. This is the primary method used for data ingestion.

A powerful conceptual bridge for students transitioning from a relational database mindset to an API-centric one is to recognize that the data returned by a REST API, while not a rigid table, can be thought of as a collection of records. For instance, a GET request to an API endpoint that returns a list of customers will typically provide a collection of JSON objects. Each JSON object, with its unique identifier and attributes such as name, address, and email, is conceptually a "row" in a "table" of customer information. By explicitly drawing this parallel, the new material feels like a natural extension of the knowledge from Week 2, preparing students for subsequent data modeling discussions where these semi-structured records will be transformed into well-defined dimension tables.[3]

1.2 Mastering requests: Your Python Gateway to APIs

The requests library is the standard for making HTTP requests in Python, abstracting away the complexities of manual web interactions. It simplifies the process of retrieving data from web APIs, making it a cornerstone of any data engineer's toolkit. The following code demonstrates its use:

Python

```python
import requests
import json

# Example 1: A simple GET request to retrieve data
# This is analogous to a 'SELECT *' from a database table.
api_url = "https://jsonplaceholder.typicode.com/posts"
response = requests.get(api_url)

# Check if the request was successful
response.raise_for_status() # Raises an HTTPError for bad responses (4xx or 5xx)

# Parse the JSON data from the response body
data = response.json()

# Example 2: A GET request with query parameters
# This is analogous to adding a 'WHERE' clause in SQL.
params = {'userId': 1, 'id': 5}
post_response = requests.get(api_url, params=params)
print("Response URL with params:", post_response.url)
print("Data for post with userId=1 and id=5:", post_response.json())

# Example 3: A POST request to create a new resource
# This is analogous to an 'INSERT INTO' statement.
new_post_data = {
    'title': 'foo',
    'body': 'bar',
    'userId': 1
}
# The 'json' parameter automatically serializes the dictionary to a JSON string
create_response = requests.post(api_url, json=new_post_data)
create_response.raise_for_status()
print("New post created:", create_response.json())
```

The requests library is powerful and intuitive. The requests.get() method is used to fetch data, while requests.post() is used to send data to the API, typically to create a new resource. The params parameter is used to add query string parameters to the URL, which can be used to filter or customize the data retrieved. This mirrors the WHERE clause in SQL. The json parameter simplifies sending a JSON payload in the request body, a common requirement for POST and PUT requests.

1.3 The Art of API Authentication

Most public and private APIs require authentication to prevent unauthorized access and manage usage. Authentication provides a secure mechanism for an application to prove its identity and gain permission to access specific data. Without it, the data pipeline would be vulnerable and non-functional.

The two most common authentication patterns are:

- **API Keys:** This is the simplest method, where a unique string (the API key) is passed with each request, typically as a query parameter or in the request header. While easy to implement, it is less secure as the key can be exposed if not handled with care.
- **OAuth 2.0:** A more robust and widely adopted standard for delegated authorization. It allows a third-party application to get a temporary "access token" to access data on behalf of a user without needing the user's password. This is commonly seen when an application asks for permission to access your social media or cloud storage data.

Regardless of the method, the data engineer must adhere to strict security practices. Hard-coding API keys directly into a script is a significant security vulnerability. Instead, keys should be stored securely in environment variables or a dedicated secrets management service. The os module in Python can be used to read these variables, ensuring sensitive information is never committed to a code repository.

1.4 Handling Reality: API Rate Limiting and Error Handling

Real-world APIs are not always perfectly reliable. They often have limitations designed to prevent abuse and ensure fair usage, a concept known as **rate limiting**. Rate limiting restricts the number of requests an application can make within a specific time window (e.g., 100 requests per minute). When a limit is exceeded, the API returns an error, commonly a 429 Too

Many Requests status code.

A production-ready data pipeline must be built with a resilient strategy for handling these errors. A robust approach involves implementing a **backoff and retry mechanism**. When a rate limit error is encountered, the script should not immediately retry the request. Instead, it should wait for a specified period before attempting again. A more advanced technique is **exponential backoff with jitter**, which involves waiting for a random duration that increases exponentially with each failed attempt. This prevents multiple parallel processes from retrying at the same time, potentially overwhelming the API and causing a cascading failure.

Furthermore, comprehensive error handling for other issues is crucial. The requests library provides requests.raise_for_status(), which automatically raises an HTTPError for a bad response (any status code from 400 to 599). This allows a data engineer to use standard try...except blocks to manage network issues, server errors, or malformed responses gracefully.

The following table summarizes common HTTP status codes and the appropriate action for a data engineer:

Status Code	Meaning	Action for a Data Engineer
200 OK	Request successful.	Continue processing the response.
400 Bad Request	The server cannot process the request due to a client error.	Log the error and stop the process for this request. Investigate the API documentation and request format.
401 Unauthorized	The request lacks valid authentication credentials.	Halt the pipeline. Check API key, token, or authentication method.
403 Forbidden	The server understands the request but refuses to authorize it.	Log the error and check permissions. This may indicate insufficient user privileges.
404 Not Found	The requested resource could not be found.	Log the error. This often means the URL is incorrect.

429 Too Many Requests	The client has sent too many requests in a given time frame.	Implement a backoff and retry strategy. Wait before attempting the request again.
500 Internal Server Error	A generic error occurred on the server.	Log the error and implement a retry mechanism. This is a server-side issue.

Section 2: Decoding the Data - The Power of JSON

2.1 The Ubiquity of JSON in the API Ecosystem

Once data is retrieved from an API, it often arrives in a format called JSON, or JavaScript Object Notation. JSON has become the de-facto standard for data exchange on the web due to its simplicity, human-readability, and lightweight nature. Its syntax is easy to read and write, and it maps directly to data structures used in modern programming languages like Python.

2.2 Anatomy of a JSON Document

JSON is built on two primary structures:

- **Objects ({}):** An unordered collection of key-value pairs. Keys must be strings, and values can be any of the JSON data types. An object is analogous to a Python dictionary.
- **Arrays (``):** An ordered list of values. An array is analogous to a Python list.

The values within a JSON object or array can be one of the following:

- A string (e.g., "Hello World")
- A number (integer or float, e.g., 42 or 3.14)
- A boolean (true or false)
- An object ({...})

- An array ([...])
- null

The ability to nest objects and arrays within one another makes JSON a powerful format for representing complex, hierarchical data, such as a customer record that includes a list of their recent orders, each with its own set of details.

The following table provides a breakdown of a sample JSON structure to help visualize these concepts:

Key	Data Type	Description/Contents
id	Number	A unique identifier for the customer.
customer_name	String	The full name of the customer.
contact_info	Object	A nested object containing contact details.
email	String	The customer's email address.
phone_number	String	The customer's phone number.
recent_orders	Array	An array of objects representing the customer's orders.
order_id	Number	A unique ID for an order within the recent_orders array.
order_date	String	The date of the order.
products	Array	An array of objects for each product in the order.

product_name	String	The name of the product.
quantity	Number	The number of units of the product sold.

2.3 Parsing JSON with Python's json Library

Python provides a built-in json module that makes working with JSON data straightforward. The most common operations are parsing a JSON string into a Python object (json.loads()) and converting a Python object into a JSON string (json.dumps()). When using the requests library, the .json() method handles the parsing step automatically, converting the JSON response body into a Python dictionary or list.

To navigate a nested JSON structure, one uses standard dictionary and list indexing, as demonstrated below:

Python

```
import json

# A sample JSON string (could be from an API response)
json_string = """
{
 "customer_id": "CUST123",
 "name": "Jane Doe",
 "contact": {
   "email": "jane.doe@example.com",
   "phone": "555-1234"
 },
 "orders":
   },
   {
    "order_id": "ORD002",
    "date": "2024-05-20",
    "items": [
     {"product": "Keyboard", "quantity": 1}
    ]
```

```
  }
 ]
}
"""

# Parse the JSON string into a Python dictionary
customer_data = json.loads(json_string)

# Accessing values
customer_name = customer_data['name']
customer_email = customer_data['contact']['email']
first_order_id = customer_data['orders']['order_id']
first_item_product = customer_data['orders']['items']['product']

print(f"Customer Name: {customer_name}")
print(f"Customer Email: {customer_email}")
print(f"First Order ID: {first_order_id}")
print(f"First Item in First Order: {first_item_product}")
```

This process of parsing and navigating the data is a crucial step in preparing it for loading into the Landing Zone. A data engineer must be able to write robust code that can handle the variability and nesting often found in JSON data from APIs. It's also important to be prepared for potential malformed data, which can raise a JSONDecodeError, and to handle this with a try...except block.

Section 3: The Second Pillar - Ingesting Raw Log Data

3.1 What are Log Files and Why are They a Valuable Data Source?

In addition to APIs, another critical source of data in a modern data platform is log files. Log files are essentially time-stamped records of events generated by systems, applications, or servers. They are the digital breadcrumbs of an application's behavior. While they may seem like simple text files, they contain a wealth of information that is invaluable for various purposes:

- **Debugging and Troubleshooting:** Log files are the first place a developer or system administrator looks to diagnose errors, trace the flow of a request, or identify

bottlenecks.
- **Security Auditing:** They can be used to track user authentication attempts, access to sensitive data, and other security-related events.
- **Business Intelligence and Monitoring:** Logs from web servers can track user clicks, page views, and time spent on a site, providing a granular view of user activity. System performance logs can reveal insights into server load, memory usage, and other operational metrics.

The true value of logs lies in their timeliness. A stream of log events, much like a stream of sensor readings, provides immediate, actionable insights.[2] For example, the Week 1 lecture materials describe a scenario where a data platform ingests real-time sensor data to detect a temperature spike, triggering an immediate alert to prevent equipment failure. This is not a theoretical exercise; it is the fundamental purpose of real-time data pipelines.[2] By understanding how to ingest and parse log data, students are taking the first step in building a system that can enable proactive monitoring and rapid response to critical business events. The act of collecting and preparing this data moves beyond a simple technical chore and becomes a core component of a system that delivers critical, real-time business value.

3.2 Common Log Formats and Their Challenges

Unlike the structured data from relational databases or the semi-structured data from a well-defined API, log data presents unique challenges. Log files are often messy, inconsistent, and lack a rigid schema. While some modern systems generate logs in a structured format like JSON, many legacy systems produce semi-structured logs with a format that is not always consistent.

Common log formats include:

- **Apache Combined Log Format:** A classic example of a semi-structured format. Each line contains a predictable but not strictly delimited set of fields, such as IP address, username, timestamp, HTTP method, and status code.
- **JSON Logs:** A more modern approach where each log line is a self-contained JSON object. This is ideal for parsing because the structure is explicit and defined.

The main challenge for the data engineer is to parse these often-unruly log files reliably. A simple string split operation may fail if a field contains a space. This is where more powerful and flexible tools are required.

3.3 Practical Log Ingestion with Python

Python provides the necessary tools for ingesting and parsing log data. A memory-efficient approach for large files is to read them line-by-line using a generator, which avoids loading the entire file into memory at once.

The re (regular expressions) module is the primary tool for parsing semi-structured log lines. A regular expression provides a pattern to match and extract specific parts of a string.

Here is a comprehensive code example demonstrating this process:

Python

```
import re

# A sample Apache combined log line
log_line = '127.0.0.1 - user-ident [10/May/2024:12:30:45 +0000] "GET /api/data/resource HTTP/1.1" 200 1234 "http://example.com/referrer" "Mozilla/5.0 (Windows NT 10.0; Win64; x64)"'

# Regular expression to match and capture parts of the log line
log_pattern = re.compile(
    r'(?P<ip_address>\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}) - '
    r'(?P<user_name>.*) '
    r'\[(?P<timestamp>.*?)\] '
    r'"(?P<method>.*?) (?P<path>.*?) (?P<http_version>.*?)" '
    r'(?P<status_code>\d+) '
    r'(?P<response_size>\d+) '
    r'"(?P<referrer>.*?)" '
    r'"(?P<user_agent>.*?)"'
)

# Search the log line for the pattern
match = log_pattern.search(log_line)

# Check if a match was found
if match:
    # Use.groupdict() to get a dictionary of named captured groups
    parsed_data = match.groupdict()
    print("Parsed Log Data:")
```

```
    for key, value in parsed_data.items():
        print(f"  {key}: {value}")
else:
    print("Could not parse the log line.")

# Example of reading a log file line-by-line
def process_log_file(filepath):
    """
    Reads a log file line by line and parses each line.
    This is memory-efficient for large files.
    """
    with open(filepath, 'r') as file:
        for line in file:
            line = line.strip()
            if line:
                match = log_pattern.search(line)
                if match:
                    yield match.groupdict()
                else:
                    print(f"Skipping unparsable line: {line}")

# A simple loop to process a hypothetical log file
# log_file_path = "server_access.log"
# for record in process_log_file(log_file_path):
#     # In a real pipeline, this record would be saved to the Landing Zone
#     print(record)
```

The use of named groups (?P<group_name>) in the regular expression makes the extracted data much more readable and organized. The re.search() method finds the first match for the pattern in a string, and match.groupdict() returns the captured groups as a dictionary, which is a perfect format for subsequent processing.

Section 4: Preserving Rawness in the Landing Zone

4.1 The Data Lake Revisited: A Quick Recap of Week 1

The fundamental concepts of a modern data platform, introduced in Week 1, are crucial for

understanding the destination of the data ingested this week. The data lake is a centralized repository that allows an organization to store all its data at any scale. It consists of several distinct zones, including the **Landing Zone**, the Staging Zone, the Integration Zone, and the Analytics Zone.[2]

This week's task—ingesting data from APIs and logs—is the crucial first step that populates the Landing Zone. It is the entry point for all data that will eventually be transformed and used for analysis.

4.2 The Mandate of the Landing Zone: Immutability and Rawness

The core principle of the Landing Zone is to serve as the "source of truth." Data is loaded into this zone in its original, raw, and immutable state. This means no transformations, no cleansing, and no changes are made to the data as it is being ingested. This principle is fundamental to a robust data platform.[3]

The rationale for this approach is multi-faceted and essential for building a resilient data pipeline:

- **Reproducibility and Auditability:** Storing raw, unchanged data provides a complete audit trail. If a transformation in a later stage is found to be flawed, the data can be re-processed from the original source without having to re-ingest it from the API or log, a process that might be difficult or impossible due to rate limits or data volume.
- **Flexibility for New Business Logic:** A business's analytical needs are not static. By preserving the raw data, new transformations can be applied in the future as business questions or requirements change.
- **Lossless Storage:** The raw format ensures that no information is lost during the ingestion process, even if the schema is complex or inconsistent.

4.3 Practical Storage Strategies for API and Log Data

While the concept of saving data to the Landing Zone is simple, a professional data engineer knows that a scalable solution requires a thoughtful storage strategy. Simply dumping files into a single directory is not a robust or efficient approach, especially for time-series data from APIs or logs.

A superior method involves partitioning the data based on a logical, time-based key. By organizing files into a directory structure like source/YYYY/MM/DD/, a data engineer makes

future processing exponentially more efficient. For example, a subsequent transformation job that needs to process all of yesterday's log data can simply point to the /logs/2024/05/20/ directory rather than scanning an entire, massive directory of all log files. This practice is an implicit requirement for a scalable data lake and directly relates to the concept of data modeling and building a continuous data pipeline.[1]

Section 5: From Ingestion to a Cohesive Multi-Source Pipeline

5.1 A Conceptual Blueprint of a Multi-Source Pipeline

The true power of this week's lesson is not just about ingesting data from APIs and logs in isolation, but in seeing how these new data sources fit into the broader vision of a multi-source data pipeline. This pipeline is the central nervous system of a modern data platform.

A conceptual flow diagram would illustrate data from various sources converging into the Landing Zone. Data from relational databases (as learned in Week 2) and the new data from APIs and log files (from this week) all feed into this raw, immutable repository. From there, the data flows into the subsequent zones for progressive refinement. It moves to the Staging Zone for initial cleansing, then to the Integration Zone for aggregation and denormalization, before finally being loaded into the Analytics Zone (often a data warehouse or data mart) for business intelligence and machine learning applications.[2]

5.2 The Chinook & Northwind Project: A Powerful Case Study

The OmniCorp project, which involves unifying data from the Chinook (music store) and Northwind (food and beverage) databases, serves as a powerful, tangible case study that bridges the gap between theory and practice.[3]

A hypothetical extension of this project provides an excellent example of a truly multi-source pipeline. The original assignment focuses on ingesting data from two relational databases, a

process that aligns perfectly with the knowledge from Week 2. However, consider a new requirement: OmniCorp wants to understand how users are interacting with their websites. This is a problem that cannot be solved with the existing relational data.

To address this, a data engineer would design a pipeline to ingest new data sources:

- **A customer feedback API:** This API returns JSON data containing customer satisfaction scores and comments. This data, a perfect candidate for ingestion using requests, would be loaded raw into the Landing Zone.
- **Web server log files:** These logs track every user click and page view. The log ingestion and parsing techniques learned this week would be used to extract meaningful events and save them to the Landing Zone.

In this scenario, the data from these disparate sources—the structured relational data from Chinook and Northwind, the semi-structured JSON from the API, and the semi-structured logs—all converge into a single, cohesive data lake. The unified DimCustomer and FactSales tables discussed in the assignment would be created in a later stage of the pipeline, combining the original sales data with new metrics derived from the web logs and customer feedback API. This process of combining disparate data sources into a unified model is the essence of a multi-source data pipeline and the ultimate goal of the DSI310 course.[3]

Section 6: Advanced Concepts & A Glimpse Forward

6.1 The Star Schema as a Knowledge Representation Framework

The concept of a Kimball Star Schema, with its central fact table and surrounding dimension tables, is the analytical backbone of the OmniCorp project.[3] However, its theoretical foundation goes deeper than simple data organization. The Week 2 lecture materials subtly connect this data modeling approach to the concepts of T-Box (Terminological Box) and A-Box (Assertion Box) from the field of Knowledge Graphs and ontology.[4]

This academic connection provides a more profound understanding of why we build fact and dimension tables the way we do. The **T-Box** represents the "terminological knowledge," which is the schema or conceptual hierarchy of the real world. In our data model, this is the role of the dimension tables. The DimCustomer table, for example, defines what a customer *is* in our business context, with attributes like CustomerName, City, and Country. The DimProduct table defines the concept of a product, including its CategoryName and GenreName.[3] These

dimensions are the permanent, descriptive attributes that provide context and define the "who, what, where, and when" of the business.

In contrast, the **A-Box** represents the "assertional knowledge," which is a collection of specific facts or events. This corresponds directly to our FactSales table.[3] A single row in this table is a specific business event, a sales transaction that happened at a certain time, involving a specific customer, employee, and product. The fact table's role is to store the transactional data, the quantitative measures of what

happened, while linking to the descriptive dimensions. This philosophical framework elevates the data modeling process from a technical exercise to an act of representing business knowledge in a structured, analytical format.

6.2 Operationalizing the Pipeline: Idempotency and Schema Evolution

Beyond the initial ingestion, a data engineer must also consider the operational challenges of maintaining a production-ready pipeline. Two key concepts are particularly relevant for data ingested from APIs and logs:

- **Idempotency:** An idempotent operation is one that can be executed multiple times without changing the result beyond the initial execution. In the context of data ingestion, a pipeline must be designed to be idempotent to prevent data duplication. For example, if a data ingestion job fails midway and is restarted, it should be able to process the same data source and only add new records without creating duplicates. This is often achieved by using a unique identifier and checking if a record already exists before inserting it.
- **Schema Evolution:** Unlike a relational database, where the schema is fixed, the schema of an API or log file can change over time. New fields may be added, a data type might change from a string to a number, or a column might be removed. A robust data pipeline must be flexible enough to handle these changes gracefully. This is another reason for the Landing Zone's "raw" approach: by storing the original data, a data engineer can handle schema changes in a later transformation step without losing the original information. This challenge is a core data engineering issue highlighted in the Chinook/Northwind assignment.[3]

Key Theories and Keywords

This section provides a formal definition of the core vocabulary introduced in this lecture.

These terms form the foundation of a modern data engineering practice.

- **API (Application Programming Interface):** A set of protocols, routines, and tools for building software applications. It defines how software components should interact.
- **Data Pipeline:** The automated process of ingesting, transforming, and loading data from one or more sources into a destination for analysis or other uses. A key learning outcome of this course is the ability to create such a pipeline.[1]
- **Data Transformation:** The process of converting data from one format or structure into another, typically to make it more usable for analysis. This step occurs after the raw data has been saved to the Landing Zone.[1]
- **Fact Table:** In a dimensional model (like a star schema), a fact table is the central table that stores the quantitative metrics (or "facts") of a business process, such as sales quantity or total revenue.[3]
- **Dimension Table:** In a dimensional model, a dimension table is a companion table to a fact table. It provides descriptive attributes for the data in the fact table, answering the "who, what, where, when, and how" of the business event.[3]
- **Idempotency:** A property of an operation that produces the same result regardless of how many times it is executed. It is a crucial concept for designing resilient and reliable data pipelines.
- **JSON (JavaScript Object Notation):** A lightweight, text-based, human-readable data interchange format used for transmitting data between a server and web application.
- **Landing Zone:** The first stage of a data lake where raw, untransformed, and immutable data is stored. It serves as the single source of truth for the data platform.[2]
- **Parsing:** The process of analyzing a string of symbols, like a log line or a JSON string, to extract meaningful components in a structured format.[1]
- **T-Box / A-Box:** Theoretical components of a knowledge base. The T-Box (Terminological Box) represents the permanent, conceptual schema (like dimension tables), while the A-Box (Assertion Box) represents specific factual assertions or events (like a fact table's rows).[4]

Works cited

1. dsi310_แบบฟอร์มเค้าโครงการบรรยาย 2025
2. dsi310 week01: Data Engineering & Data Lake
3. dsi310: Data Lake Modeling
4. 2025 dsi310 week02