International Journal of AI, Big Data, Computational and Management Studies
Noble Scholar Research Group | Volume 7, Issue 1, 233-237, 2026
ISSN: 3050-9416 | https://doi.org/10.63282/3050-9416.IJAIBDCMS-V7I1P135

Original Article

Python automation of Broadband Subscriber data integration to SaaS

Amey Deshpande
Calix Inc,McKinney, USA.

Received On: *16/01/2025* ***Revised On:*** *18/01/2026* ***Accepted On:*** *24/02/2026* ***Published On:*** *08/03/2026*

Abstract: *Software as a Service (SaaS) providers commonly develop and offer application programming interface (API) resources that enable businesses to seamlessly integrate their operational and customer data. These cloud-based applications are especially valuable to Broadband Service Providers (BSPs), who rely on product APIs to build customized solutions tailored to their organizational needs. Within these SaaS platforms, various types of data—such as access network metrics, subscriber billing records, and layer-3 routed Netflows information—are aggregated to deliver comprehensive functionality. Despite these advantages, the integration of such diverse datasets often presents significant challenges in terms of data quality. Data sourced from multiple systems may contain numerous errors, omissions, and inconsistencies, which can result in unreliable insights and hinder the effectiveness of the application. It is imperative that all data undergoes cleaning and standardization prior to integration. Data exploration and transformation remain a challenging prerequisite to the application of data analysis methods. The desired transformations are often ad-hoc so that existing end-user tools may not suffice, and plain programming may be necessary [3]. This ensures that the information conforms to established frameworks and specifications, thereby facilitating the successful adoption and optimal performance of the SaaS product within the BSP's operational environment.*

Keywords: *Python, Libraries, Saas, API, JSON.*

1. Introduction

A BSP in the Midwest had implemented a SaaS product where several elements of their network data came together to be viewed and used by the customer support representatives (CSR). The SaaS product hosted on cloud, was a point of convergence for subscriber demographic information, customer premises equipment (CPE) or router, and the IP flows captured from the BSP's core routers. The CPE devices had the ability to call home and do periodic TR-069 informs to the product and stay checked in. The IP flows were captured as samples from the BSP's core routers and required a one-time setup. The most critical data, however, was subscriber data. This data was dynamically changing every day and contained a large number of attributes that tied the subscriber to its services and CPE. This data was integrated to the SaaS product by a CSR manually in the product's User Interface (UI). Any new subscribers turned up with broadband service, or existing subscribers getting an update to their services were captured by the CSR manually in the product UI. This framework of data integration was neither reliable due to human error possibilities, nor was it an efficient use of the CSRs. The BSP had plans to integrate data over the open access API resources of the cloud native product to benefit from real time and automated data modifications to the SaaS product.

2. Saas Requirements

A SaaS product is designed to provide a support, marketing and operations environment to the BSP. To adopt a product for any of these purposes, the BSP is required to bring in necessary data to integrate in either a cadence or in a real time setting. BSPs typically rely on Business Support Systems (BSS) providers and Operations Support Systems (OSS) providers to track daily billing activities along with network and field operations that go in alignment with the billing system. The SaaS platform brought together data across all platforms used by the BSP, layered with its own analytics and integration giving a seamless presentation of the whole business. The products required three sets of data:

2.1. Billing data

This data included all information related to the subscriber and was unique to their account with the BSP.

- Subscriber name, address, phone, email, etc.
- Service and subscription codes
- Device identifiers
- Other billing attributes like start date, special services, etc.

2.2. Access Network data

This data included information regarding all equipment involved in the access network from the Central Office (CO) to the last mile of the subscriber.

- Layer-2 Ethernet and Gigabit Passive Optical Network (GPON) switches.
- Optical Network Terminal (ONT).
- Customer Premises Equipment (CPE).
- Node – Shelf – Card – Port identifiers.

2.3. Layer-3 routing data

This data included information pertaining the layer-3 routed traffic for the subscribers of this BSP:

- IP Netflows imported from the core or edge routers in the BSP's network.
- Remote Authentication Dial-in User Service (RADIUS) identifiers for Point to Point over Ethernet (PPPoE) subscribers imported from a Radius server.
- Internet Protocol version 4 (IPv4) subnets defined for capturing and mapping IP traffic limited to the BSP.

3. Data Convergence

3.1. Framework

To deliver services using the BSP's data, the SaaS platform required an architecture that consolidated data from multiple sources into a centralized cloud database, or otherwise known as the data lake. The amount of data being collected daily was massive. This data can be used to gain insights and make informed decisions. The process of using the data to gain insights is becoming essential in the modern world but along with the power of making intelligent and informed decisions by using data there comes a challenge of making this process cost effective and fast. Tasks like data integration and data transformation become very complex when we are dealing with large databases. Naturally nontechnical people face issues while accessing the database let alone attempting to integrate and transform [2]. The figure below represents multiple sources of the data converging via the correlation engine and BSS integration to the SaaS platform presented on the application layer to the BSP via a cloud native product UI.

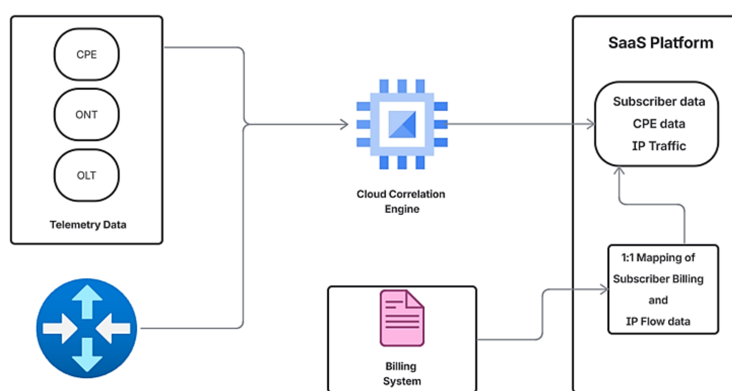

Fig 1: High Level BSP Data Convergence to the Saas Platform

3.2. Cloud Architecture

There are various moving parts to a cloud engine especially when data extraction is involved. In today's data-driven world, organizations face the challenge of managing and harnessing vast amounts of data from various sources. This is where data lakes come into play. A data lake is a centralized repository that stores structured, semi-structured, and unstructured data in its raw format [1]. Within this architecture, certain data types, such as telemetry data from variety of hardware in the access network, were stored in the data lake in their raw format with minimal structuring. On the contrary, subscriber data sourced from the billing system required transformation into a predefined framework before being written to the cloud database. Given that millions of subscribers from multiple BSPs accessed a shared SaaS platform, the varying syntax and headers from each data source necessitated standardized structuring prior to database entry.

3. Python

BSS providers developed to the API resources made available by the SaaS platform; however, the framework could not involve updating all existing data on the product. The development of the BSS was limited to capitalize on new and upcoming billing data only. The goal to fill the large gap of data integration between a billing system and the SaaS product gave birth to python enabled automation on the platform side. This meant that the data team on the SaaS platform created an automated option to get raw data from billing systems, transform it into a usable framework and then query API resources in iterative fashion.

The python-based script was written to perform a global refresh of all existing data that was originally input manually by CSRs from the BSP side. This was a broad solution assimilating data formats like CSV and JSON on two ends of data transformation. It also supported API resources to be queried within the script along with attributes like rate limiting and logging. All these features of the automation were stood up based on leveraging multiple libraries and functions written as a python code.

3.1. Requests

This is a fundamental, yet effective HTTP library. 'Requests' allows you to send HTTP/1.1 requests extremely easily. There's no need to manually add query strings to your URLs, or to form-encode your PUT & POST data [4]. In this python script not only was the 'requests' library imported, but also underlying functions such as the adapter named 'HTTPAdapter' and 'Retry' function from "requests.packages.urllib3.util". Requests was used for 3 main functions in the python script

- Capture responses when an API resource on the cloud platform was queried.
- Capture any exceptions or irregular respones when APIs were called.
- 'requests.Session' configured with a retry strategy to handle transient network errors and specific HTTP status codes (500, 502, 504).

The syntax for querying the SaaS product's API endpoints was like the below:

```
r = requests.post(token_url, headers=headers, data=data)

if r.status_code == 200:
return r.json(), last_request_time
```

3.2. JSON:

The acronym JSON stands for JavaScript Object Notation. JSON is a language agnostic format for presenting and writing data and recognized as a standard for data interchange. JSON was essentially the transformed or queried data at the end of the script that was executed. The API reponses that contained the payload related to subscriber billing data were in the JSON standard. One primary function of importing the json library into the python code was to reference the credentials required for accessing the Api resources that were stored in the directory in a api_credentials.json file. The syntax to reference the credentials within a script was as below:

```
with open("api_credentials.json", 'r', encoding='utf-8') as json_file:
        credentials = json.load(json_file)
```

When any API resource is queried, especially with a GET request, the server tends to respond with a HTTP code 200 response and the following data payload is in the JOSN format. This payload could be a huge data set pertaining to what API was queried. The json library in python was supportive of capturing and writing this payload as the output of the query. The RFC requires that JSON be represented using either UTF-8, UTF-16, or UTF-32, with UTF-8 being the recommended default for maximum interoperability. As permitted, though not required, by the RFC, this module's serializer sets ensure_ascii=True by default, thus escaping the output so that the resulting strings only contain printable ASCII characters. Other than the ensure_ascii parameter, this module is defined strictly in terms of conversion between Python objects and Unicode strings, and thus does not otherwise directly address the issue of character encodings[5]. The syntax for such a python code is as below:

```
if response.status_code == 200:
    return response.json(), last_request_time
else:
error_json = response.json()
error_code = error_json.get('errorCode', 'Unknown')
    error_message = error_json.get('errorMessage', 'Unknown')
```

3.3. CSV

The CSV format is a very common import and export format for spreadsheets and databases. CSV format was used for many years prior to attempts to describe the format in a standardized way in RFC 4180. The lack of a well-defined standard means that subtle differences often exist in the data produced and consumed by different applications. These differences can make it annoying to process CSV files from multiple sources. Still, while the delimiters and quoting characters vary, the overall format is similar enough that it is possible to write a single module which can efficiently manipulate such data, hiding the details of reading and writing the data from the programmer.

The csv module implements classes to read and write tabular data in CSV format. It allows programmers to say, "write this data in the format preferred by Excel," or "read data from this file which was generated by Excel," without knowing the precise details of the CSV format used by Excel. Programmers can also describe the CSV formats understood by other applications or define their own special-purpose CSV formats.

The csv module's reader and writer objects read and write sequences. Programmers can also read and write data in dictionary form using the DictReader and DictWriter classes[6]. The syntax for reading from a csv file and loading it into a python dictionary is as below:

```
with open(csv_file_path, mode='r') as file:
 reader = csv.DictReader(file)
     for row in reader:
         subscriber_data = {
        <broadband subscriber payload>
             }
```

Referencing and reading from a .csv file can be handled either by using a strong library like *pandas* within the python script and use dataframes to load the csv data into. Another option is to link a user interface using the python library *tkinter* to have the user the ease of referencing the csv file through a user interface dialog. The syntax for such a python code is below:

```
root = tk.Tk()
root.withdraw()
csv_file_path = filedialog.askopenfilename(title="Select a CSV file", filetypes=(("CSV files", "*.csv"), ("All files", "*.*")))
root.destroy()
if csv_file_path:  # Proceed only if a file was selected
    try:
 process_subscribers(clientId, token, csv_path, rate_limit_ms, request_time, expires_at, client_secret, token_grant['refresh_token'], orgId)
except Exception as e:
     logger.error(f"An error occurred during subscriber processing: {e}")
    else:
     logger.info("CSV file was not selected by user.")
```

4. Algorithm and implementation

4.1. Algorithm Outline

4.1.1. Initialize and Load State

- Load configuration like endpoints, scopes, timeouts and rate limits.
- Load last successful checkpoint timestamp and offset.

4.1.2. Authenticate

- Obtain API access token (OAuth2 client credentials, signed JWT, or API key—depending on SaaS).
- Prepare standard headers and correlation identifiers.

4.1.3. Read Data

- Query the source of truth, which is the CSV file containing a whole database of subscriber, service and equipment data.

4.1.4. Transform & Validate

- Normalize formats like dates and upper/lower casing.
- Validate required fields and enforce constraints which would be the unique subscriber key.
- Map all attributes in the CSV file to match with the SaaS API schema.
- Ensure a one-to-one relationship between the subscriber and associating fields to avoid duplicates.

4.1.5. Write Data

- If CREATE: call POST /subscribers
- If UPDATE: call PATCH/PUT /subscribers/{id}
- If NO-OP: skip
- Services: Add service attributes and associate to subscriber.
- CPE: Associate device identifiers to subscriber profile or relationship endpoint.

4.1.6. Handle Failures and Retries

- For transient HTTP error like 429 or 503/504 timeouts: exponential backoff with jitter.
- For auth errors like 401 or403: refresh token and retry once.

4.1.7. Implementation Flow

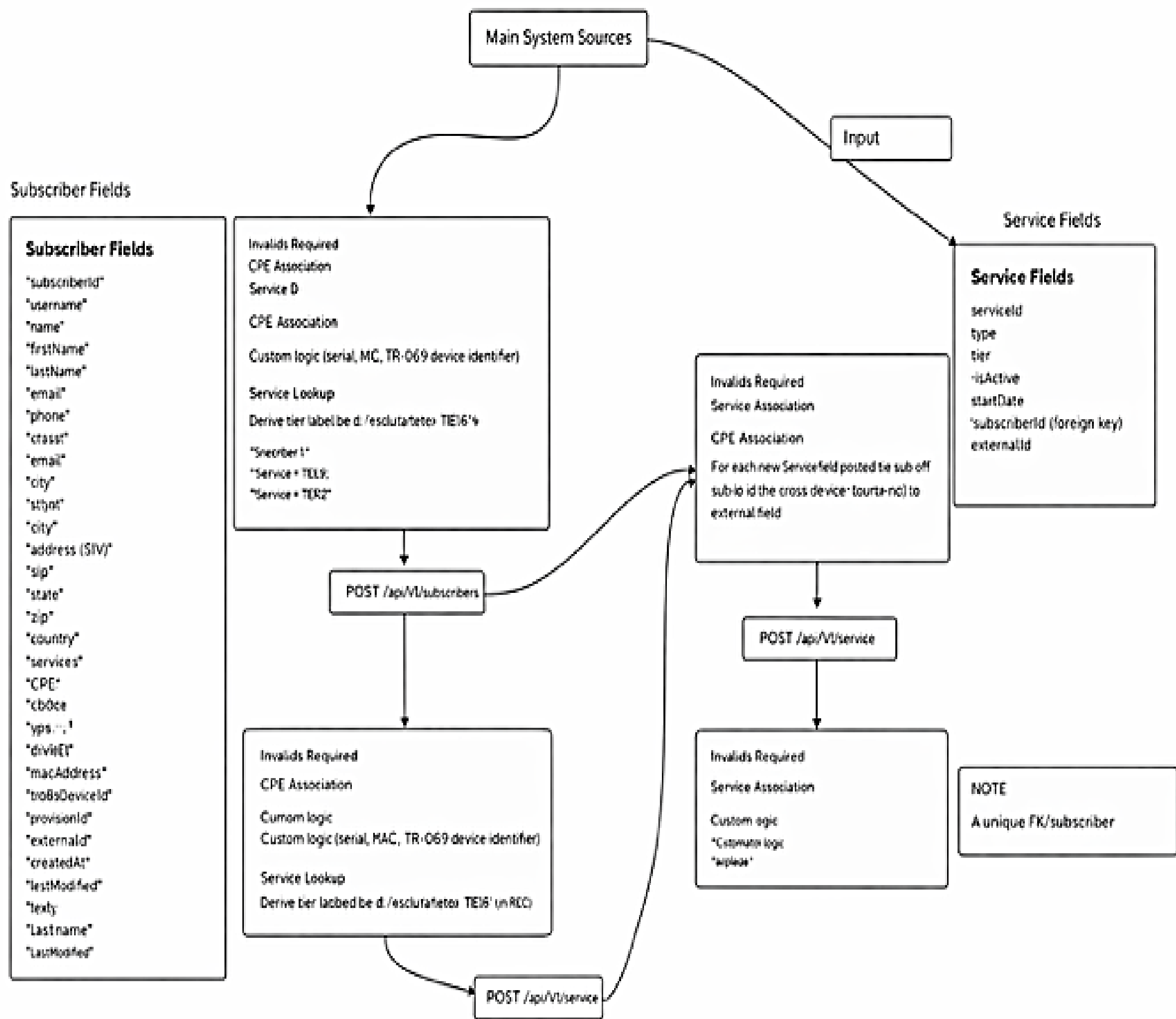

Fig 2: Subscriber and Service Data Flow with CPE Association and API Integration

5. Conclusion

As APIs exist everywhere, they are an abundant source of external and internal data for a BSP and their SaaS product. Having a powerful ecosystem of libraries and tools, Python provides a strong platform for solving these problems. The Python ecosystem is rich — you won't be missing a library to do anything that you want in the data pipeline from fetching and cleaning data to visualizing it. Moreover, a real-time, asynchronous programming is becoming a trend, because of how critical systems need fast decision making. It also covers important challenges such as authentication, rate limiting, error handling, ensuring that data pipelines are reliable but also resilient [7]. This research demonstrates that effective SaaS adoption within a Broadband Service Provider environment depends fundamentally on disciplined data preparation and integration practices. As subscriber and network data originates from heterogeneous systems with varying quality and structure, systematic cleaning, validation, and standardization are essential to mitigate errors and inconsistencies that undermine reliability. The findings underscore that ad-hoc transformations and domain-specific logic often exceed the capabilities of generic end-user tools, necessitating programmable integration approaches to ensure conformance with established schemas and operational requirements. By enforcing these data engineering principles prior to and during integration, BSPs can achieve more accurate, timely, and scalable data synchronization, thereby improving operational efficiency, reducing manual intervention, and enabling the SaaS platform to deliver its intended value with consistency and confidence.

References

1. Building a Data Lake: A Step-by-Step Guide with Codes and Examples, by Pradyumna Karkhane
2. Data Integration and Transformation using Large Language Models Anu Mohan M, Dr. Ashok K, Muskan Rizwan Shaikh, Dhanush Y P and Yajat Vishwakarma
3. Dexteris: Data Exploration and Transformation with a Guided Que Builder Approach Sébastien Ferré⋆ Univ Rennes, CNRS, Inria, IRISA F-35000 Rennes, France
4. https://pypi.org/project/requests/
5. https://docs.python.org/3/library/json.html
6. https://docs.python.org/3/library/csv.html
7. A Pythonic Approach to API Data Management: Fetching, Processing, and Displaying Data for Business Intelligence by Divya Kodi - Cyber Security Senior Data Analyst, Department of Cyber Security, Truist Financial, CA, USA.
8. Agarwal, S. (2023). Multi-Modal Deep Learning for Unified Search-Recommendation Systems in Hybrid Content Platforms. International Journal of AI, BigData, Computational and Management Studies, 4(3), 30-39. https://doi.org/10.63282/3050-9416.IJAIBDCMS-V4I3P104.