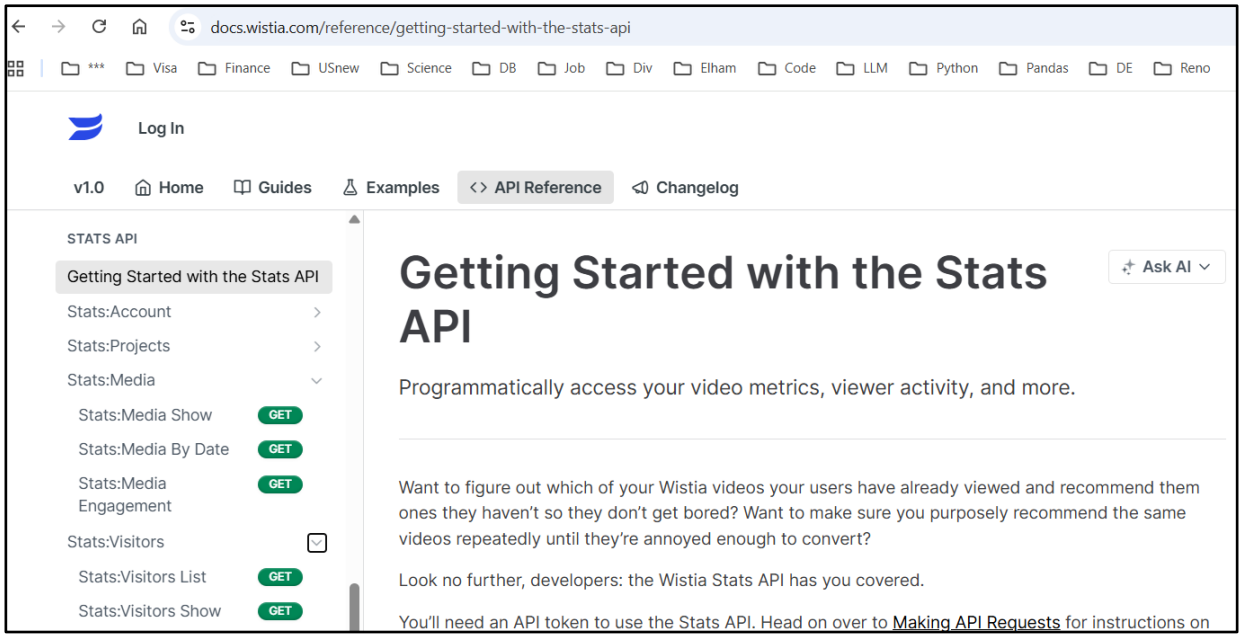
Project 4: Wistia Video Analytics

Overview

In this porject, we will work with JSON format data, and the goal is to pull them from the Wistia Portal. Before, Desiging the data model pipeline, I wanted to explore on the Wistia Portal, download the data to my local machine and explore on the data briefly, to understand them well. After that I will focus on the data model structure while focussing on the AWS cloud services.

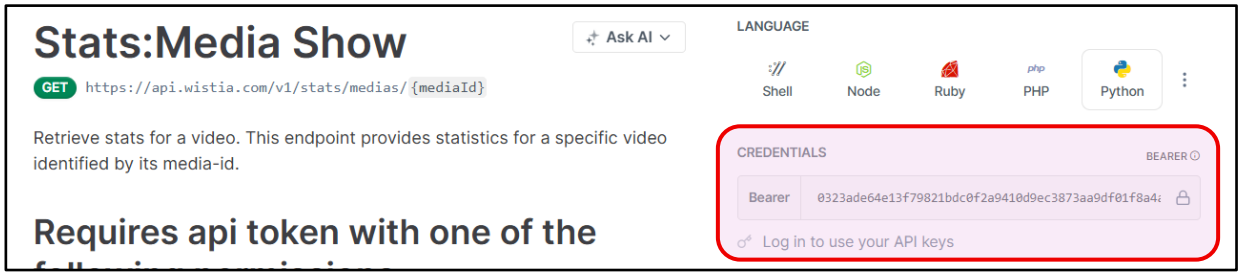
Wistia API Exploration

In the Wistia API portal, https://docs.wistia.com/reference/getting-started-with-the-stats-api , there is a section, named STATS API, which we will focus on that side.



Authentication

To access the data of Wistia API, I need a secure API Token. I put it in the **Bearer** section of **Credentials**.



Endpoints:

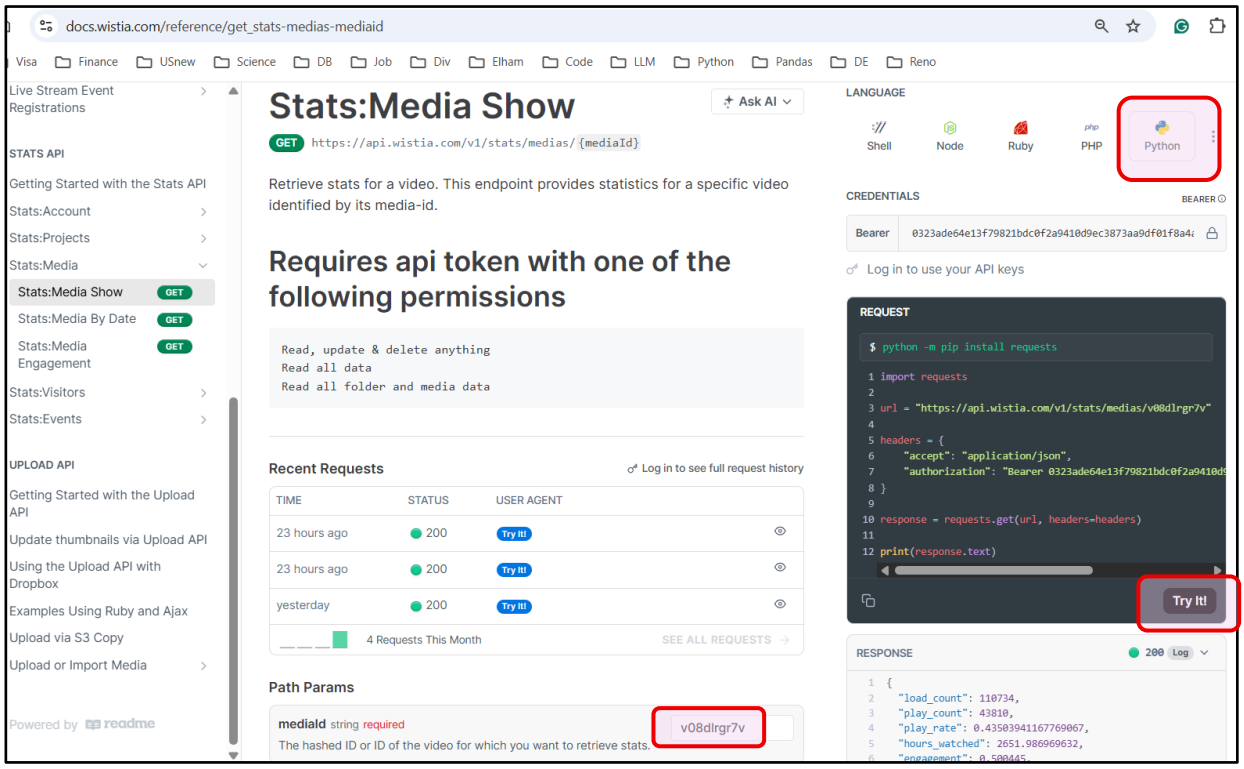
As the project requested, I will focus on two parts of **Stats:Media** and **Stats:Visitors**. In the **Stats:Media**, there are two useful endpoints, and in the **Stats:Visitors**, there are one useful endpoint, which I will use all of them.

1)	**Stats:Media Show**	`Get` `https://api.wistia.com/v1/stats/medias/{mediaId}`
2)	**Stats:Media Engagement**	`Get` `https://api.wistia.com/v1/stats/medias/{mediaId}/engagement`
3)	**Stats:Visitors List**	`Get` `https://api.wistia.com/v1/stats/visitors`

Get information

Based on projdect description, I have to focus on only two media with `mediaId` of **"v08dlrgr7v"** and **"gskhw4w4lm"**. (I think one of them is related to FaceBook and the other is related to YouTube).

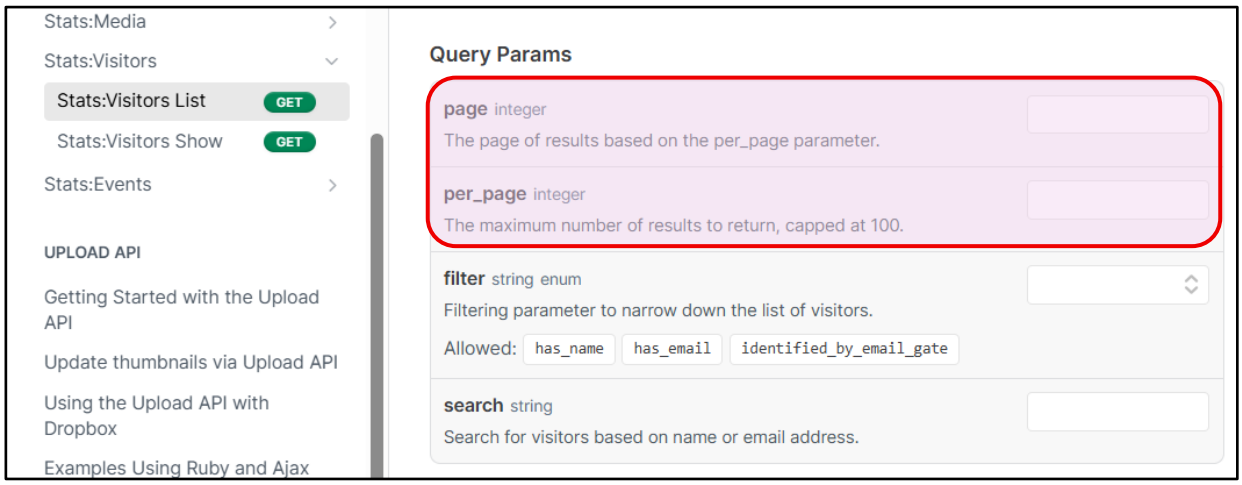
To get the information about these two media, I need to put each of those two `mediaIds` in the parameter section of the first endpoint and then considert the **Python language** and click on the **Try It!** button, as below image.



For example, the `mediaId` of **"v08dlrgr7v"**, shows that this media loaded 110734 times by 94642 different visitors. Also, the `mediaId` of **"gskhw4w4lm"**, shows that this media loaded 111165 times by 104555 different visitors.

Also, we can use the second endpoint, **Stats:Visitors List**, to find detailed information aobut users' engagement by selected media; the time of play and pouse of the media by users.

Moreover, at the visitor level, we can use the third endpoint, **Stats:Media Engagement**, to find general information about total 110361 visitors, who engaging with Wistia. This level of information not connected directly with media-level information. Therefore, in the parameters section, we cannot put the `mediaId`. However, as the number records are very huge, 110361, then to get specific information, we can use other filter parameters, like **page**, **per_page**, and **search**. It means that we need to **pagination** during getting this information. The maximum records per_page are capped at **100**.



This is information on one of those **110361 visitors**:


```
[
  {
    "visitor_key": "1767733_916148e7-3375-4eed-9e82-0ba235e2155a-c87496ba3-abbd20671cf2-6259",
    "created_at": "2026-01-06T21:03:42.000Z",
    "last_active_at": "2026-01-06T21:03:24.000Z",
    "last_event_key": "1767733_21b83d0a-18fb-4699-81f1-ae70d3abfe37-bb91c8ef9-8079955f2fa3-80ff",
    "load_count": 1,
    "play_count": 1,
    "identifying_event_key": null,
    "visitor_identity": {
      "name": "",
      "email": null,
```

```
      "org": {
        "name": null,
        "title": null
      }
    },
    "user_agent_details": {
      "browser": "Instagram",
      "browser_version": "410",
      "platform": "iOS (iPhone)",
      "mobile": true
    }
  }
]
```


To extract all of these information, I have created two Python files, **client.py**, for configuration, and **ingestion.py**, for ingestion the data. If I need the incremental data, I have to repeat running the **ingestion.py** file, like daily! This file, downloaded all files related to only two media and also all visitors connected to Wistia.

Folder Structure

I considered and created this folder structure in my local laptop (data folder, the blue ones, will be created after running the **ingestion.py** file):

```
wistia/
├── config/
│   └── wistia_config.yaml
├── src/
│   ├── client.py
│   └── ingestion.py
└── data/
    ├── medias/
    ├── media_engagement/
    └── visitors/
```

I used YAML file for configuration of data ingestion. In the **config/wistia_config.yaml** file I put some parameters and medias' name:

yaml file:

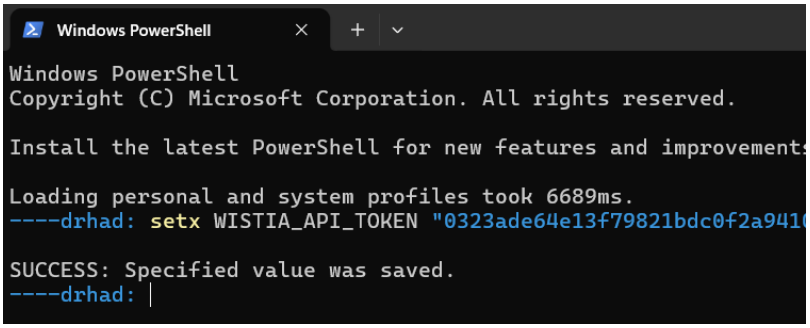
```yaml
api:
  base_url: https://api.wistia.com/v1
  per_page: 100
  timeout: 50

medias:
  - gskhw4w4lm
  - v08dlrgr7v
```

Set Token in my laptop

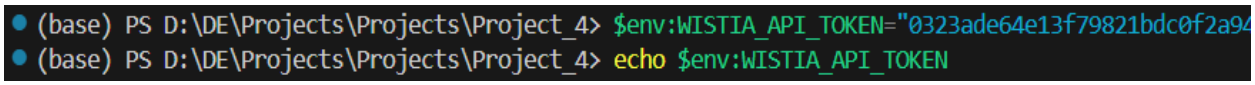
To connect Windows (PowerShell) to Wistia, for secure networking and data extraction, I need to set Environment Variable, locally and save the Token in it:

```
setx WISTIA_API_TOKEN "0323ade64e13f****"
```



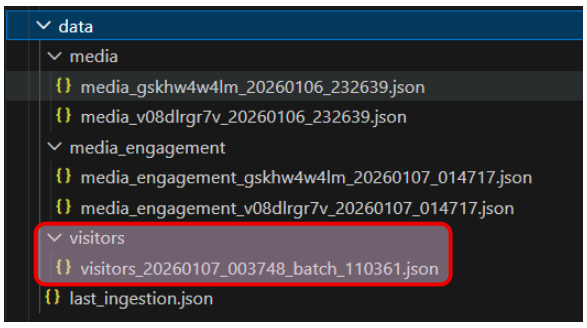
As I am workig mostly with VS Code, I did similar action for VS Code and set my Wistia Token as Environment Variable locally like this:

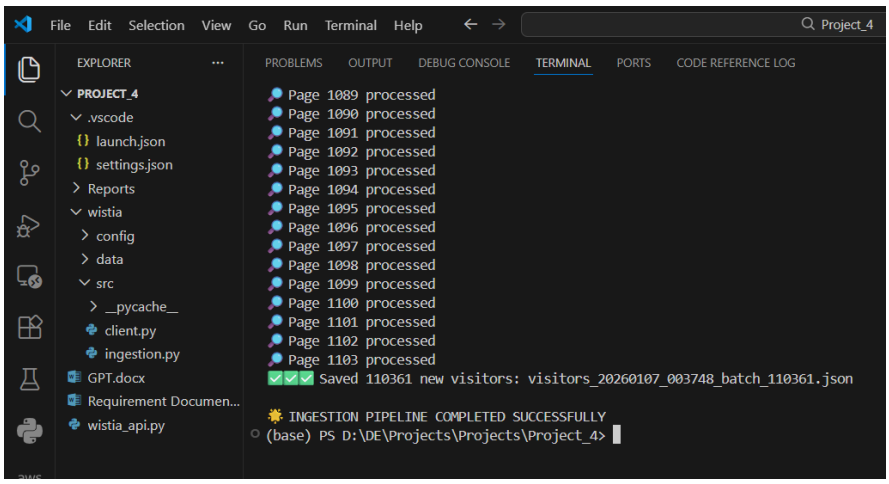
```
$env:WISTIA_API_TOKEN="0323ade64e13f****"
```



Data Ingestion 1st Run

To ingest the data, I ran the **ingestion.py** file successfully and then got the data folder structure. There are two JSON files inside both **media** and **media_engagement** folders, and one JSON file inside **visitors** folder. This **visitors_20260107_003748_batch_110361.json** file consists of information of total **110361 visitors**.

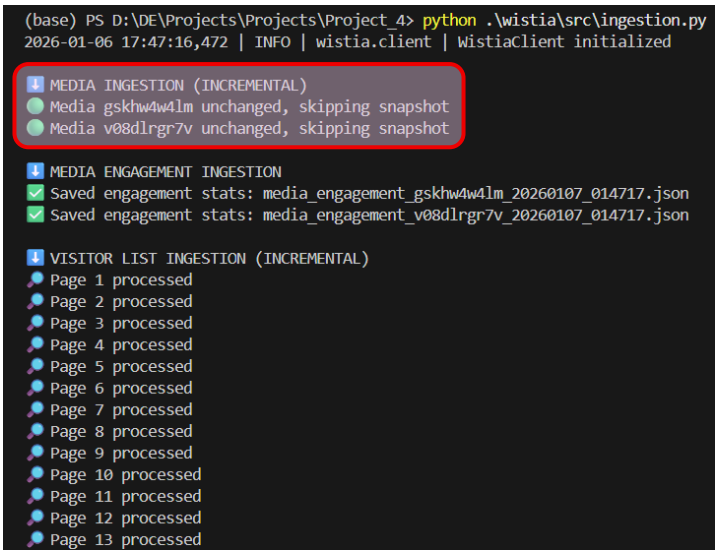


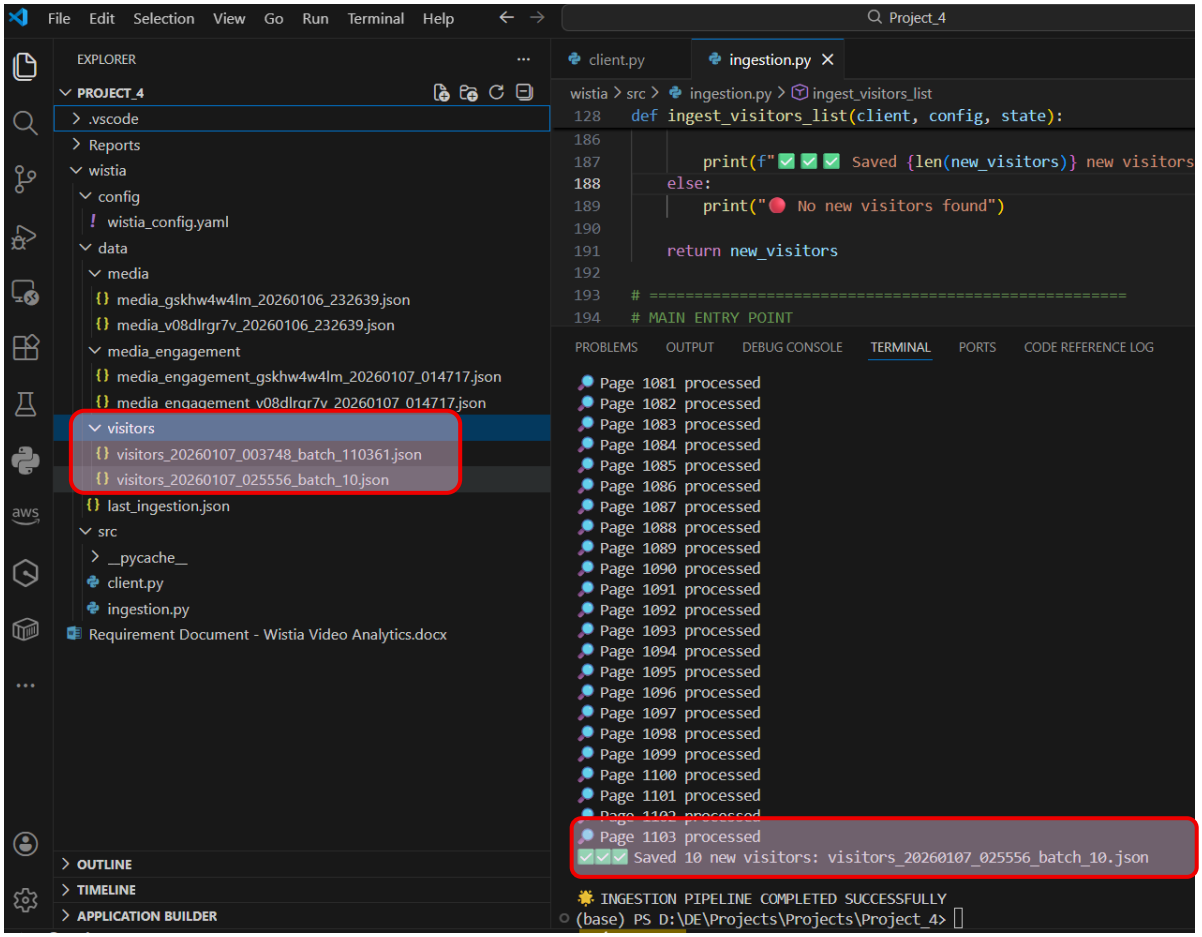


Data Ingestion 2nd Run

To test the incremental data fetching, I have run the **ingestion.py** code manually after about **2 hours** for the second time. I found that the **media-level** information is not changed. As you can see in the figure below, inside the red triangle, both media-level files are unchanged.

However, after finishig running the **ingestion.py** code, I noticed that during about 2 hourse, there are some updates in the **visitor-level**. There were 10 new visitors, and the new information is saved in new JSON file **visitors_20260107_025556_batch_10.json**. As you can see, in the primary visitors file **visitors_20260107_003748_batch_110361.json**, which saved at the time of 00:37:48, there are 110361 visitors, however in the second file, **visitors_20260107_025556_batch_10.json**, which saved at the time of 02:55:56, there are only 10 new visitors. Compare the information inside the red triangulars in the figure below.





Data Ingestion 3rd Run

Again, to test the incremental data fetching, I have run the **ingestion.py** code manually after **1 day** for the third time. I found that the **media-level** information is changed. As you can see in the figure below, inside the red triangle, both media-level files are changed. For example, as I look at to information of **media_Id "gskhw4w4lm"**, I found that the **in 1 day, the laoding time is icreased by 1 using 1 new visitor.**

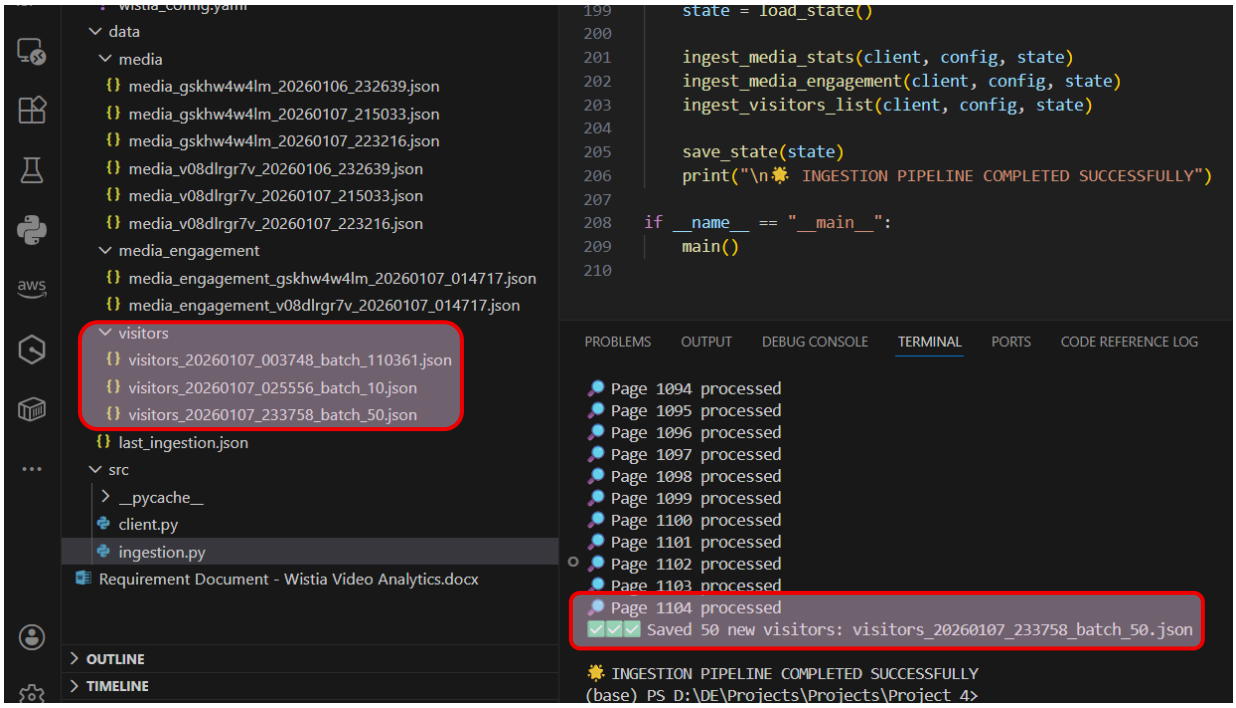
media_gskhw4w4lm_20260106_232639.json

```
{
  "load_count": 111165,
  "play_count": 16681,
  "play_rate": 0.15495193917077135,
  "hours_watched": 336.97566205632,
  "engagement": 0.167007,
  "visitors": 104555
}
```

media_gskhw4w4lm_20260107_215033.json

```
{
  "load_count": 111166,
  "play_count": 16681,
  "play_rate": 0.15495045717127665,
  "hours_watched": 336.97566205632,
  "engagement": 0.167007,
  "visitors": 104556
}
```

Also, for **1 day**, there are some updates in the **visitor-level** too. There are **50 new visitors**, which their information is saved in new JSON file **visitors_20260107_233758_batch_50.json**. As you can see, in the primary visitors file **visitors_20260107_003748_batch_110361.json**, which saved at the time of 00:37:48, there are 110361 visitors, however in the third file, **visitors_20260107_233758_batch_50.json**, which saved at the time of 23:37:58, there are only 50 new visitors. Compare the information inside the red triangulars in the figure below.



It means that the incremental data are fetched successfully!

Data Structure

After downloading all data (1st run), now I have three different dictinary files in the JSON format, two on **media-level** and one on **visitor-level**.

Medias

dim_media_v media_Id = v08dlrgr7v	
load_count	110734
play_count	43810
play_rate	0.43503941167769067
hours_watched	2651.986969632
engagement	0.500445
visitors	94642

dim_media_g media_Id = gskhw4w4lm	
load_count	111165
play_count	16681
play_rate	0.15495193917077135
hours_watched	336.97566205632
engagement	0.167007
visitors	104555

Media Engagements

media_Id = v08dlrgr7v	
engagement	0.500445
engagement_data	[45074, 40660, …]

media_Id = gskhw4w4lm	
engagement	0.167007
engagement_data	[16108, 9796, …]

Visitors

Visitors		
visitor_key		
created_at		
last_active_at		
last_event_key		
load_count		
play_count		
identifying_event_key		
visitor_identity	name	
	email	
	org	name
		title
user_agent_details	browser	
	browser_version	
	platform	
	mobile	

After getting the approval from SME, my plan is to ingest these data into (AWS S3) Bronze layer, then flatten them into rational data format and clean them in the (AWS S3) Silver layer and prepare other BI-bsed dim and fact tables in the (AWS S3) Gold layer for visualization, ML and BI analysis.

Proposal:

After completing exploratory data analysis on the three source datasets, the following proposal outlines the design and implementation of a production-ready data pipeline to support business insights, customer analytics, and reporting requirements.

Step 1 – Data Ingestion (S3 Bronze Layer)

Objective

The objective of this step is to ingest raw transactional and dimensional data from a Wistia API into an Amazon S3–based data lake, forming the Bronze layer. This layer will store raw, immutable data to support auditability, reprocessing, and downstream transformations. I will schedule daily running the data ingestion, to fetch the incremental data for one week.

Design Choice

To comply with the project requirement that allows me to use Python for data ingestion, I will implement **AWS Glue Python Shell Job** for data ingestion instead of AWS Lambda, to have orchestration and consistency with Glue Workflow. This approach ensures that the ingestion process is both scalable and aligned with other AWS Glue ETL Jobs in Transformation layers.

Security and Connectivity

The following AWS services will be used to enable secure and reliable ingestion:

- **AWS Secrets Manager:** Stores Wistia API credentials securely and prevents hardcoding sensitive information in Glue job script.
- **AWS IAM:** Enforces least-privilege access policies, allowing Glue jobs to read secrets and write data to Amazon S3 securely.

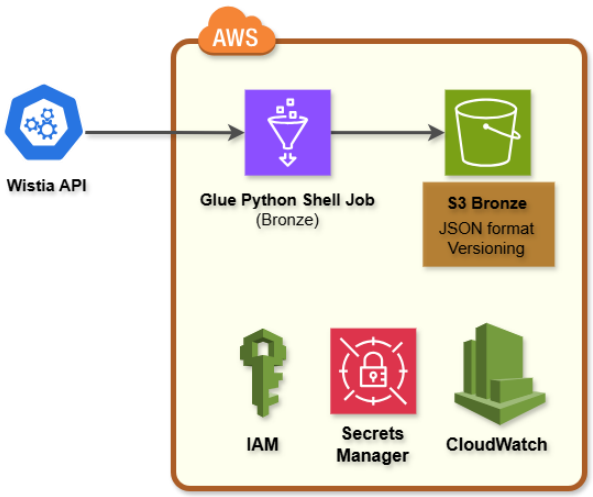
Target Storage (Bronze Layer)

Raw data will be written separately to the following S3 bucket: **s3://project4-bronze-bucket/**

```
Project4-bronze-bucket/
├── raw-data/
│   ├── media/
│   │   └── media_<media_id>_<timestamp>.json
│   │
│   ├── media_engagement/
│   │   └── media_engagement_<media_id>_<timestamp>.json
│   │
│   ├── visitors/
│   │   └── visitors_batch_<count>_at_<timestamp>.json
│
├── state/
│   └── last_ingestion.json
```

Implementation Approach

- AWS Glue Python Shell job will connect to Wistia API.
- Source JSON files (**media-level**, and **visitor-level**) will be extracted in batch mode.
- Data will be written on Amazon S3 in **JSON** format.
- No transformations will be applied at this stage to preserve data fidelity.
- Enable S3 bucket versioning to retain all versions of the raw JSON files and keep incremental data for 7 days.



Step 2 – Transformation (S3 Silver Layer)

Objective

The objective of the Silver layer is to transform raw ingested JSON files into **clean**, **standardized**, and **analytically usable datasets**. This step focuses on improving data quality by resolving **missing values**, **removing duplicates**, **enforcing schemas**, and **preparing fact and dimension tables** for downstream analytics and metric computation.

Design and Storage

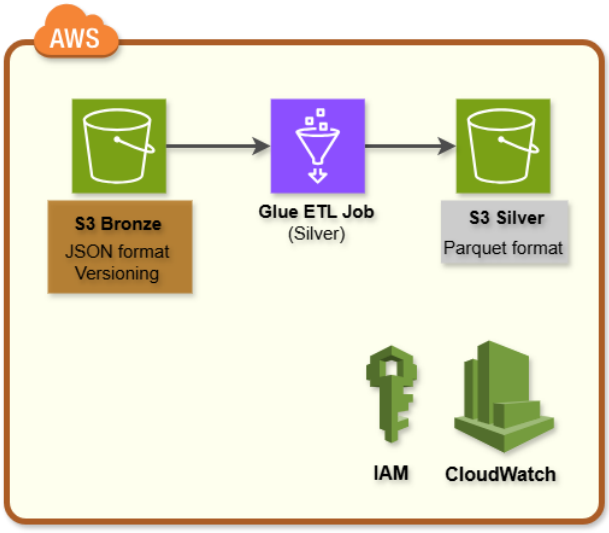
Data in this layer will be structured into fact and dimension–oriented datasets, enabling efficient joins and aggregations in subsequent processing stages. Transformed data will be stored in a dedicated S3 bucket: **s3://project4-silver-bucket/**

```
Project4-silver-bucket/
├── fact_media/
│   └── media_stats.parquet
│
├── fact_media_engagement/
│   └── media_engagement.parquet
│
├── fact_visitors/
│   └── visitors.parquet
```

Implementation Approach

An AWS Glue PySpark ETL job, named **project4-silver-glue**, will be executed to perform the following transformations:

- **Schema Enforcement**
  - Convert JSON files into rational datasets amd define schemas for fact and dimension tables to prevent schema drift.
  - Ensure consistent column ordering and naming conventions.
- **Data Type Standardization**
  - Cast columns to appropriate data types (e.g., timestamps, numeric fields, boolean flags).
  - Ensure consistency across datasets for join keys and metrics.
- **Duplicate Handling**
  - Identify and remove duplicate records based on defined primary or composite keys.
  - Preserve only the most recent or valid records where applicable.
- **Missing Value Management**
  - Remove records with critical missing identifiers (e.g., primary keys).
  - Retain non-critical nulls where they are analytically meaningful (e.g., visitor_key).
- **Column Normalization**
  - Standardize column names (e.g., lowercase, snake_case) for consistency and usability.
  - Align naming conventions across all datasets.
- **Optimized Storage Format**
  - Persist transformed datasets in **Parquet format** to improve query performance and reduce storage costs.



Step 3 – Analytics & Aggregation (S3 Gold Layer)

Objective

In this step, the objective is to produce analytics-ready, aggregated datasets by joining and summarizing the dimensional and fact tables created in the Silver layer. The Gold layer serves as the single source of truth for business metrics and is optimized for direct consumption by visualization and reporting tools.

Design and Storage

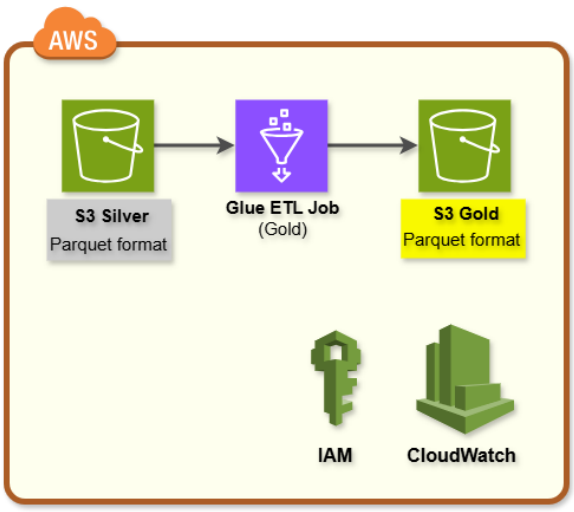
Data in this layer will be structured into different fact and dimension datasets. Transformed data will be stored in a dedicated S3 bucket: **s3://project4-gold-bucket/**

```
project4-gold-bucket/
├── fact_media_performance/
│   └── daily_media_metrics.parquet
│
├── fact_audience_insights/
│   └── visitor_metrics.parquet
```

Implementation Approach

One AWS Glue PySpark ETL job, named **project4-gold-glue** (for dimension-level and fact-level aggregations and KPI calculations), will generate curated datasets and metrics aligned with the business analytics questions, including but not limited to:

- Statistical metrics regarding each media
- Engagement distrbution for each media
- Play Trends over Time
- Classification of each media based on play_count, play_rate, and hours_watched
- Classification of visitors based on browser-level, platform-level, and mobile-level
- RF (Recency, Frequency) snapshot of visitors



Step 4 – Visualization (Streamlit)

Objective

In the final step, Streamlit dashboards will query data directly from the Gold S3 bucket.

Implementation Approach

- Using pandas for data handling (sufficient for moderate-sized parquet files).
- Build interactive dashboards for stakeholders with filters, charts, and KPI cards.

Note: No data warehouse or query engine is required; this approach is completely serverless and cost-efficient.

Step 5 – Jobs Orchestration and CI/CD

Objective

The objective of this component is to automate the end-to-end batch data pipeline on a daily schedule while ensuring reliable orchestration, clear job dependencies, and maintainable CI/CD practices. All data is ingested from SQL Server into AWS, transformed through Bronze, Silver, and Gold layers, and made available for analytics and visualization without manual intervention. To support continuous development and deployment, all AWS Glue Spark scripts are version-controlled in a GitHub repository and automatically deployed to AWS when changes occur.

Scheduling and Orchestration

- A daily schedule glue jobs using **AWS Glue Trigger**.
- The **Glue Workflow** acts as the central orchestration engine for the pipeline.
- The Glue Workflow executes jobs in a dependency-based sequence:

Glue Python Shell Job (Data Ingestion) → Glue Silver ETL Job → Gold Glue Jobs

CI/CD Strategy

This approach enables automated deployment, traceability, and reproducibility of data pipelines.

- All Python and PySpark scripts for Glue jobs are stored in a **GitHub** repository**:** https://github.com/Hadi2468/ELT_Project4.
- A CI/CD pipeline using **AWS CodePipeline** automatically:
  - Detects changes or additions to Glue job scripts
  - Deploys updated scripts to an S3 code bucket: **s3://project4-code-bucket/**
- All Glue jobs reference their scripts directly from the S3 code bucket, ensuring:
  - Version consistency
  - Easy rollback
  - No manual code uploads

Data Ingestion (Bronze Layer)

- The **Glue Python Shell job** is triggered by the **Glue Workflow**.
- It connects securely to Wistia API and ingests raw JSON files.
- The ingested data is written to the **S3 Bronze bucket** without transformation.

Silver Layer Transformation

- Upon successful completion of the ingestion step, the Glue Workflow triggers the **Silver Glue ETL job**.
- This job:
  - Cleans and standardizes the raw data
  - Resolves missing values and duplicates
  - Produces normalized dimension and fact tables
- Outputs are stored in the **S3 Silver layer** in Parquet format.

Gold Layer Aggregation

- After the Silver job completes successfully, the Glue Workflow triggers **Gold Glue ETL job**.
- This job:
  - Compute KPIs and analytical aggregates
  - Produce dashboard-ready datasets
- Final outputs are stored in the **S3 Gold layer**.

Monitoring and Failure Handling

- Each Glue job writes logs to **Amazon CloudWatch**.
- In case of any job failure:
  - The Glue Workflow stops downstream execution
  - Error details are available in CloudWatch Logs for troubleshooting

Streamlit Visualization

- Streamlit dashboards read Gold-layer Parquet files directly from S3.
- Dashboards always reflect the latest successful pipeline execution.
- No additional transformation logic is required at the visualization layer.

Benefits of This Orchestration Strategy

This design ensures:

- Clear and deterministic job dependencies
- Controlled execution order
- Parallel processing where appropriate
- Failure isolation and observability
- Minimal operational complexity
- Scalable and production-ready CI/CD practices

Conclusion

This project successfully demonstrates the end-to-end design of a scalable, production-ready data analytics pipeline for Wistia video engagement data, through the Wistia Stats API: media-level, media engagement–level, and visitor-level datasets.

The proposed architecture follows modern data lake best practices by implementing a multi-layered Bronze–Silver–Gold design on Amazon S3. The Bronze layer preserves raw, immutable JSON data for auditability and reprocessing, the Silver layer standardizes and normalizes the data into analytics-ready Parquet formats, and the Gold layer delivers curated fact tables and KPIs optimized for business intelligence, reporting, and advanced analytics. This layered approach ensures data quality, scalability, and long-term maintainability.

AWS Glue was intentionally selected as the core orchestration and transformation engine to provide consistency across ingestion and transformation stages, native scheduling, dependency management, and seamless integration with AWS IAM, Secrets Manager, and CloudWatch. The use of Glue Workflows enables deterministic execution order, clear failure handling, and operational observability, while avoiding unnecessary architectural complexity. Incremental ingestion logic and state management further enhance efficiency by minimizing redundant data processing.

The pipeline is fully automated and extensible, supporting manual execution, scheduled daily runs, and CI/CD-driven deployments through GitHub and AWS CodePipeline. This ensures that changes to ingestion or transformation logic can be deployed safely, reproducibly, and with minimal operational overhead. The final Streamlit-based visualization layer provides a lightweight, serverless interface for stakeholders to explore engagement metrics and audience insights directly from the Gold layer.

Overall, this proposal delivers a robust, cost-efficient, and enterprise-aligned analytics solution that transforms raw Wistia API data into actionable insights. The design is intentionally modular, secure, and scalable, making it well-suited for future extensions such as additional media sources, longer retention periods, advanced audience segmentation, and machine learning–based engagement analysis.