Kafka Topics Partition Count Recommender Application

> **TL;DR:** ***End Kafka performance headaches.*** *This smart recommender reads your historical consumption data and delivers precise partition recommendations that optimize throughput and enable effortless scaling—no more over-provisioning or under-utilizing your topics.*

The **Kafka Cluster Topics Partition Count Recommender Application** offers data-driven accuracy for Kafka topic sizing. By analyzing past consumption trends, that is, the average consumption records in bytes, it uses this information to determine consumer throughput. Then, over a rolling n-day period, it identifies the average consumption of records in bytes, scaling that number by a factor of X to forecast future demand and calculate the required throughput. Next, it divides the required throughput by the consumer throughput and rounds the result to the nearest whole number to determine the optimal number of partitions. The result is an intelligent, automated recommendation system that ensures each Kafka topic has the appropriate number of partitions to handle current workload and support future growth effectively.

Table of Contents

1.0 To get started

Download ---> ***Configure*** ---> ***Run*** ---> ***Results***

1.1 Download the Application

Clone the repo: `shell git clone https://github.com/j3-signalroom/kafka_cluster-topics-partition_count_recommender-app.git`

Since this project was built using `uv`, please install it, and then run the following command to install all the project dependencies:

```
uv sync
```

1.1.1 Special Note on two custom dependencies

This project has *two custom dependencies* that we want to bring to your attention:

1. `cc-clients-python_lib`: *This library offers a simple way to interact with Confluent Cloud services, including the Metrics API. It makes it easier to send API requests and manage responses. It is used in this project to connect to the Confluent Cloud Metrics API and retrieve topic consumption metrics.*
2. `aws-clients-python_lib`: *This library is used to interact with AWS services, specifically AWS Secrets Manager in this case. It enables the application to securely retrieve secrets stored in AWS Secrets Manager.*

1.2 Configure the Application

Now, you need to set up the application by creating a `.env` file in the root directory of your project. This file will store all the essential environment variables required for the application to connect to your Confluent Cloud Kafka cluster and function correctly. Additionally, you can choose to use **AWS Secrets Manager** to manage your secrets.

> **Note**: *Your Confluent Cloud API Key, Secret, and Kafka Cluster ID are required to access the Confluent Cloud Metrics API and retrieve topic metrics. Additionally, your Bootstrap Server URI, along with your Kafka API Key and Secret, are necessary to access the designated Kafka Cluster.*

1.2.1 Create the `.env` file

Create the `.env` file and add the following environment variables, filling them with your Confluent Cloud credentials and other required values:

```
# Environment variables credentials for Confluent Cloud and Kafka clusters
CONFLUENT_CLOUD_CREDENTIAL={"confluent_cloud_api_key":"<YOUR_CONFLUENT_CLOUD_API_KEY>", "confluent_cloud_api_secret":
"<YOUR_CONFLUENT_CLOUD_API_SECRETS>"}
KAFKA_CREDENTIALS=[{"kafka_cluster_id": "<YOUR_KAFKA_CLUSTER_ID>", "bootstrap.servers": "
<YOUR_BOOTSTRAP_SERVER_URI>", "sasl.username": "<YOUR_KAFKA_API_KEY>", "sasl.password": "<YOUR_KAFKA_API_SECRET>"}]

# AWS Secrets Manager Secrets for Confluent Cloud and Kafka clusters
USE_AWS_SECRETS_MANAGER=<True|False>
CONFLUENT_CLOUD_API_SECRET_PATH={"region_name": "<YOUR_SECRET_AWS_REGION_NAME>", "secret_name": "
<YOUR_CONFLUENT_CLOUD_API_KEY_AWS_SECRETS>"}
KAFKA_API_SECRET_PATHS=[{"region_name": "<YOUR_SECRET_AWS_REGION_NAME>", "secret_name": "
<YOUR_KAFKA_API_KEY_AWS_SECRETS>"}]

# Topic analysis configuration
INCLUDE_INTERNAL_TOPICS=<True|False>
TOPIC_FILTER=<YOUR_TOPIC_FILTER, IF ANY>

# Throughput and partition calculation configuration
REQUIRED_CONSUMPTION_THROUGHPUT_FACTOR=<YOUR_REQUIRED_CONSUMPTION_THROUGHPUT_FACTOR>

# Sampling configuration
USE_SAMPLE_RECORDS=<True|False>
SAMPLING_DAYS=<YOUR_SAMPLING_DAYS>
SAMPLING_BATCH_SIZE=<YOUR_SAMPLING_BATCH_SIZE>
```

The environment variables are defined as follows:

Environment Variable Name	Type	Description	Example	Default	Requ
`CONFLUENT_CLOUD_CREDENTIAL`	JSON Object	Contains authentication credentials for Confluent Cloud API access. Must include `confluent_cloud_api_key` and `confluent_cloud_api_secret` fields for authenticating with Confluent Cloud services.	`{"confluent_cloud_api_key": "CKABCD123456", "confluent_cloud_api_secret": "xyz789secretkey"}`	None	Yes (i Mana
`KAFKA_CREDENTIALS`	JSON Array	Array of Kafka cluster connection objects. Each object must contain `sasl.username`, `sasl.password`, `kafka_cluster_id`, and `bootstrap.servers` for connecting to specific Kafka clusters.	`[{"sasl.username": "ABC123", "sasl.password": "secret123", "kafka_cluster_id": "lkc-abc123", "bootstrap.servers": "pkc-123.us-east-1.aws.confluent.cloud:9092"}]`	None	Yes (i Mana
`USE_AWS_SECRETS_MANAGER`	Boolean	Controls whether to retrieve credentials from AWS Secrets Manager instead of using direct environment variables. When `True`, credentials are fetched from AWS Secrets Manager using the paths specified in other variables.	`True` or `False`	`False`	No

Environment Variable Name	Type	Description	Example	Default	Requ
`CONFLUENT_CLOUD_API_SECRET_PATH`	JSON Object	AWS Secrets Manager configuration for Confluent Cloud credentials. Contains `region_name` (AWS region) and `secret_name` (name of the secret in AWS Secrets Manager). Only used when `USE_AWS_SECRETS_MANAGER` is `True`.	`{"region_name": "us-east-1", "secret_name": "confluent-cloud-api-credentials"}`	None	Yes (i `USE_` is `Tru`
`KAFKA_API_SECRET_PATHS`	JSON Array	Array of AWS Secrets Manager configurations for Kafka cluster credentials. Each object contains `region_name` and `secret_name` for retrieving cluster-specific credentials from AWS Secrets Manager.	`[{"region_name": "us-east-1", "secret_name": "kafka-cluster-1-creds"}, {"region_name": "us-east-1", "secret_name": "kafka-cluster-2-creds"}]`	None	Yes (i `USE_` is `Tru`
`INCLUDE_INTERNAL_TOPICS`	Boolean	Determines whether Kafka internal topics (system topics like `__consumer_offsets`, `_schemas`) are included in the analysis and reporting. Set to `False` to exclude internal topics and focus only on user-created topics.	`True` or `False`	`False`	No
`TOPIC_FILTER`	Comma-separated String	A list of specific topic names or part of topic names to analyze. When provided, only these topics will be included in the analysis. Use commas to separate multiple topic names. Leave blank or unset to analyze all available topics.	`"user-events,order-processing,payment-notifications"`	Empty (all topics)	No
`REQUIRED_CONSUMPTION_THROUGHPUT_FACTOR`	Float/Integer	Multiplier applied to current peak consumption rates for capacity planning and future demand forecasting. A value of `3` means planning for 3x the current peak throughput (300% of current load).	`3` (for 300%), `2.5` (for 250%)	`3`	No
`USE_SAMPLE_RECORDS`	Boolean	Enables record sampling mode for analysis instead of processing all records. When `True`, only a subset of records is analyzed for performance optimization. Recommended for large topics or initial analysis.	`True` or `False`	`True`	No
`SAMPLING_BATCH_SIZE`	Integer	Maximum number of records to sample per topic when `USE_SAMPLE_RECORDS` is `True`. Controls the sample size for analysis to balance accuracy with performance. Larger values provide more accurate analysis but slower processing.	`50000`, `100000`	`50000`	No
`SAMPLING_DAYS`	Integer	Time window (in days) for record sampling, creating a rolling window that looks back from the current time. Defines how far back to sample records for analysis. **Note**: Topics with retention periods shorter than this value will use their maximum available retention period instead.	`7` (last week), `30` (last month)	`7`	No

1.2.2 Using the AWS Secrets Manager (optional)

If you use **AWS Secrets Manager** to manage your secrets, set the `USE_AWS_SECRETS_MANAGER` variable to `True` and the application will retrieve the secrets from AWS Secrets Manager using the names provided in `CONFLUENT_CLOUD_API_KEY_AWS_SECRETS` and `KAFKA_API_KEY_AWS_SECRETS`.

The code expects the `CONFLUENT_CLOUD_API_KEY_AWS_SECRETS` to be stored in JSON format with these keys:

- `confluent_cloud_api_key`
- `confluent_cloud_api_secret`

The code expects the `KAFKA_API_KEY_AWS_SECRETS` to be stored in JSON format with these keys:

- `kafka_cluster_id`
- `bootstrap.servers`
- `sasl.username`
- `sasl.password`

1.3 Run the Application

Navigate to the Project Root Directory

Open your Terminal and navigate to the root folder of the `kafka_cluster-topics-partition_count_recommender-app/` repository that you have cloned. You can do this by executing:

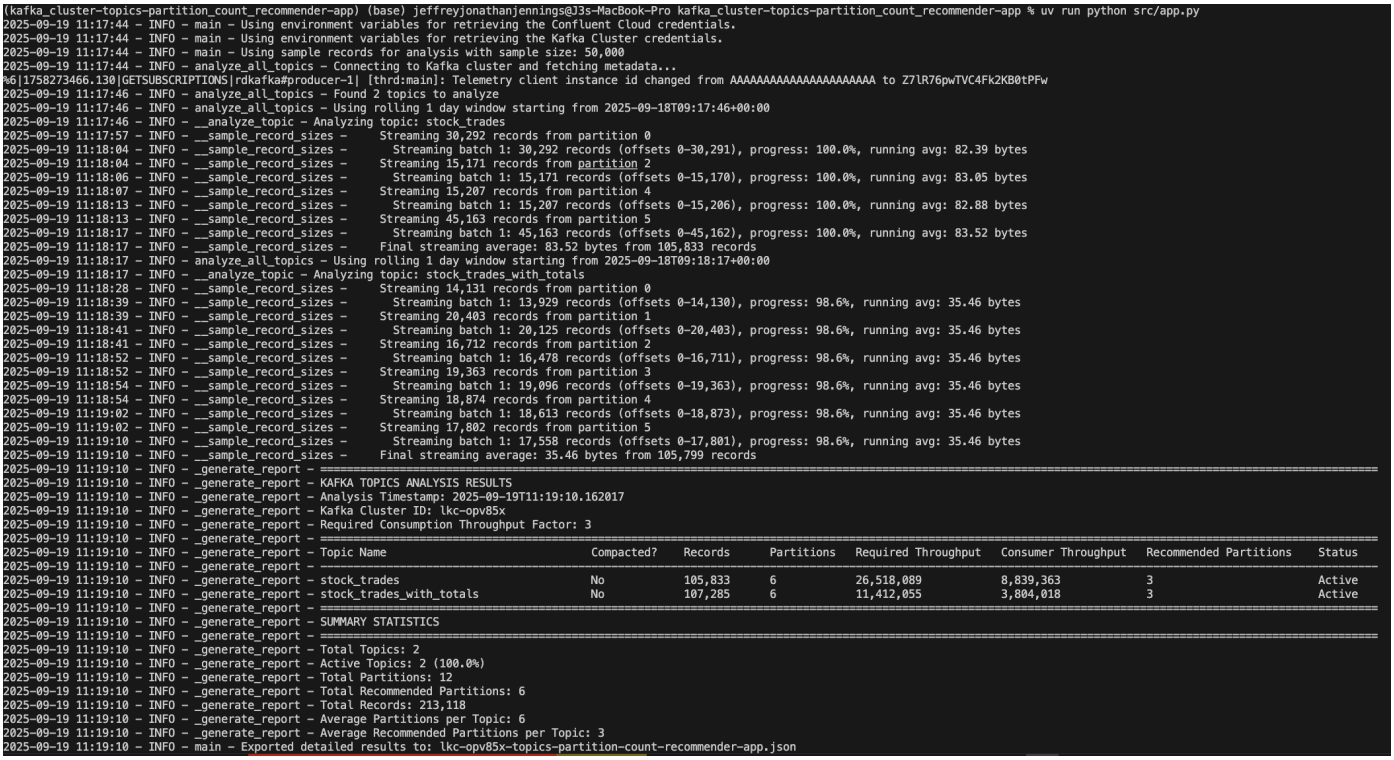
```
cd path/to/kafka_cluster-topics-partition_count_recommender-app/
```

> Replace `path/to/` with the actual path where your repository is located.

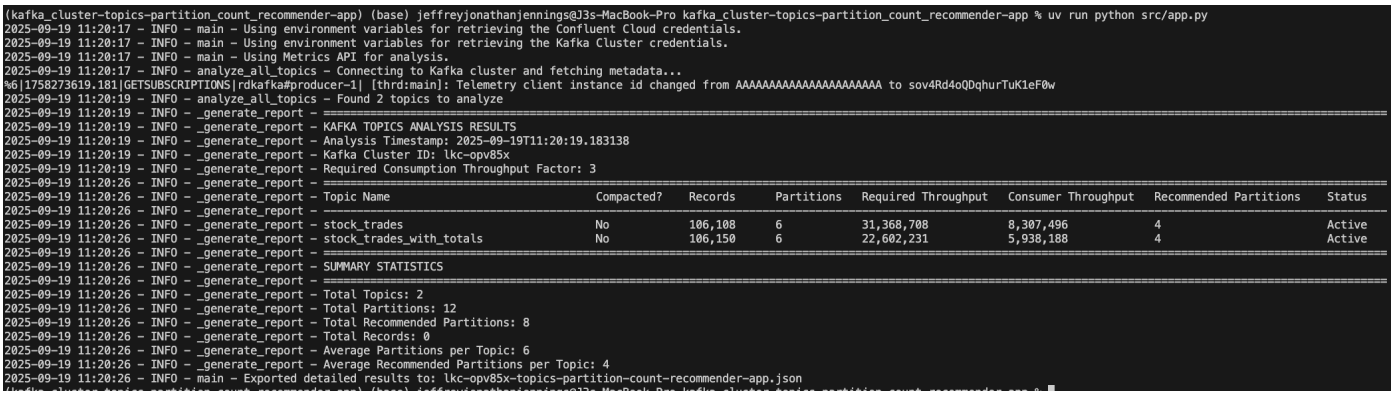
Then enter the following command below to run the application:

```
uv run python src/app.py
```

If `USE_SAMPLE_RECORDS` environment variable is set to `True`, the application will sample records from each topic to calculate the average record size in bytes. For example, below is a screenshot of the application running successfully:



If `USE_SAMPLE_RECORDS` is set to `False`, the application will use the Confluent Cloud Metrics API to retrieve the average and peak consumption in bytes over a rolling seven-day period. For example, below is a screenshot of the application running successfully:



1.3.1 Did you notice we prefix `uv run` to `python src/app.py`?

You maybe asking yourself why. Well, `uv` is an incredibly fast Python package installer and dependency resolver, written in **Rust**, and designed to seamlessly replace `pip`, `pipx`, `poetry`, `pyenv`, `twine`, `virtualenv`, and more in your workflows. By prefixing `uv run` to a command, you're ensuring that the command runs in an optimal Python environment.

Now, let's go a little deeper into the magic behind `uv run`:

- When you use it with a file ending in `.py` or an HTTP(S) URL, `uv` treats it as a script and runs it with a Python interpreter. In other words, `uv run file.py` is equivalent to `uv run python file.py`. If you're working with a URL, `uv` even downloads it temporarily to execute it. Any inline dependency metadata is installed into an isolated, temporary environment—meaning zero leftover mess! When used with `-`, the input will be read from `stdin`, and treated as a Python script.
- If used in a project directory, `uv` will automatically create or update the project environment before running the command.
- Outside of a project, if there's a virtual environment present in your current directory (or any parent directory), `uv` runs the command in that environment. If no environment is found, it uses the interpreter's environment.

So what does this mean when we put `uv run` before `python src/app.py`? It means `uv` takes care of all the setup—fast and seamless—right in your local environment. If you think AI/ML is magic, the work the folks at Astral have done with `uv` is pure wizardry!

Curious to learn more about Astral's `uv`? Check these out:

- Documentation: Learn about `uv`.
- Video: `uv` IS THE FUTURE OF PYTHON PACKING!.

If you have Kafka connectivity issues, you can verify connectivity using the following command:

1.3.2 Troubleshoot Connectivity Issues (if any)

To verify connectivity to your Kafka cluster, you can use the `kafka-topics.sh` command-line tool. First, download the Kafka binaries from the Apache Kafka website and extract them. Navigate to the `bin` directory of the extracted Kafka folder. Second, create a `client.properties` file with your Kafka credentials:

```
# For SASL_SSL (most common for cloud services)
security.protocol=SASL_SSL
sasl.mechanism=PLAIN
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required \
  username="<YOUR_KAFKA_API_KEY>" \
  password="<YOUR_KAFKA_API_SECRET>";

# Additional SSL settings if needed
ssl.endpoint.identification.algorithm=https
```

Finally, run the following command to list all topics in your Kafka cluster:

```
./kafka-topics.sh --list --bootstrap-server <YOUR_BOOTSTRAP_SERVER_URI> --command-config ./client.properties
```

If the connection is successful, you should see a list of topics in your Kafka cluster. If you encounter any errors, double-check your credentials and network connectivity.

1.4 The Results

When the application has completed iterating through the list of topics, it will produce a .CSV file in the root folder of the application. The CSV file contains for each topic, the computed average consumer throughput in MB/s, required throughput in MB/s, which is use to calculate the recommended partition count for the topic. Below is an example of a CSV output:

> The name of the CSV comprises of the `<KAFKA CLUSTER ID>-recommender-<CURRENT EPOCH TIME IN SECONDS>-report.csv`. The current epoch time is to help with making the file name unique.

2.0 How the app calculates the recommended partition count

The app uses the Kafka `AdminClient` to retrieve all Kafka Topics (based on the `TOPIC_FILTER` specified) stored in your Kafka Cluster, including the original partition count per topic. Then, it iterates through each Kafka Topic, calling the Confluent Cloud Metrics RESTful API to retrieve the topic's average (i.e., the *Consumer Throughput*) and peak consumption in bytes over a rolling seven-day period. Next, it calculates the required throughput by multiplying the peak consumption by the `REQUIRED_CONSUMPTION_THROUGHPUT_FACTOR` (i.e., the *Required Throughput*). Finally, it divides the required throughput by the consumer throughput and rounds the result to the nearest whole number to determine the optimal number of partitions.

> **Note**: *This why the app requires the Kafka API Key and Secret to connect to your Kafka Cluster via the AdminClient, and the Confluent Cloud API Key and Secret to connect to the Confluent Cloud Metrics API.*

For example, suppose you have a consumer that consumes at **25MB/s**, but the the consumer requirement is a throughput of **1.22GB/s**. How many partitions should you have?

To determine the number of partitions needed to support a throughput of **1.22GB/s** for a Kafka consumer that can only consume at **25MB/s**, you can calculate it as follows:

1. Convert the target throughput to the same units:
    - **1.22GB/s = 1250MB/s**
2. Divide the target throughput by the consumer's capacity:

$$\textbf{Partition Count} = \frac{\textit{Required Throughput}}{\textit{Consumer Throughput}} = \frac{1250\ MB/s}{25\ MB/s} = \mathbf{50}$$

3. Since you can only have a whole number of partitions, you should always round up to the nearest whole number:

$$\textbf{Partition Count} = 50$$

The **50 partitions** ensure that the consumer can achieve the required throughput of **1.22GB/s** while consuming at a rate of **25MB/s** per partition. This will allow the workload to be distributed across partitions so that multiple consumers can work in parallel to meet the throughput requirement.

2.1 End-to-End Application Workflow

```
---
id: 3eadd090-8b11-4a52-abc2-9755066ffc9d
---
sequenceDiagram
    participant User
    participant App as app.py
    participant KTA as KafkaTopicsAnalyzer
    participant AC as AdminClient
    participant MC as MetricsClient
    participant KC as Consumer
    participant AWS as AWS Secrets Manager
    participant CSV as CSV Report

    User->>App: Run application
    App->>App: Load environment variables
    App->>App: Read configuration settings

    alt Use AWS Secrets Manager
        App->>AWS: Get Confluent Cloud credentials
        AWS-->>App: Return credentials
        App->>AWS: Get Kafka API credentials
        AWS-->>App: Return Kafka credentials
    else Use Environment Variables
        App->>App: Read credentials from env vars
    end

    loop For each Kafka cluster
        App->>KTA: Initialize(cluster_id, bootstrap_server, api_key, api_secret, metrics_config)
        KTA->>AC: Create AdminClient
        KTA->>MC: Create MetricsClient

        App->>KTA: analyze_all_topics(parameters)

        KTA->>AC: list_topics()
        AC-->>KTA: Return topics metadata

        KTA->>AC: describe_configs(topics)
        AC-->>KTA: Return topic configurations

        KTA->>KTA: Filter topics (internal, topic_filter)
```

```
        KTA->>CSV: Create report file with headers

        loop For each topic to analyze
            alt Use Sample Records
                rect rgb(173, 216, 230)
                    KTA->>KTA: Calculate rolling window timeframe
                    KTA->>KTA: __analyze_topic()

                    KTA->>KC: Create Consumer
                    KTA->>KC: get_watermark_offsets()
                    KC-->>KTA: Return low/high offsets

                    KTA->>KC: offsets_for_times()
                    KC-->>KTA: Return offset at timestamp

                    KTA->>KTA: __sample_record_sizes()

                    loop For each partition
                        KTA->>KC: assign() and seek()

                        loop Sample records in batches
                            KTA->>KC: poll()
                            KC-->>KTA: Return record
                            KTA->>KTA: Calculate record size
                            KTA->>KTA: Update running totals
                        end
                    end

                    KTA->>KC: close()
                    KTA->>KTA: Calculate average record size
                end

            else Use Metrics API
                rect rgb(255, 182, 193)
                    KTA->>MC: get_topic_daily_aggregated_totals(RECEIVED_BYTES)
                    MC-->>KTA: Return bytes metrics

                    KTA->>MC: get_topic_daily_aggregated_totals(RECEIVED_RECORDS)
                    MC-->>KTA: Return records metrics

                    KTA->>KTA: Calculate average bytes per record
                end
            end

            KTA->>KTA: Calculate required throughput
            KTA->>KTA: Determine recommended partitions
            KTA->>CSV: Write topic analysis to report
        end

        KTA->>KTA: Generate summary statistics
        KTA-->>App: Return analysis results
    end

    App->>User: Display completion message
    App->>CSV: Save report file
```

3.0 Unlocking High-Performance Consumer Throughput

The throughput of a **Kafka consumer** refers to the rate at which it can read data from Kafka topics, typically measured in terms of **megabytes per second (MB/s)** or **records per second**. Consumer throughput depends on several factors, including the configuration of Kafka, the consumer application, and the underlying infrastructure.

3.1 Key Factors Affecting Consumer Throughput

3.1.1 Partitions

- Throughput scales with the number of partitions assigned to the consumer. A consumer can read from multiple partitions concurrently, but the total throughput is bounded by the number of partitions and their data production rates.
- Increasing the number of partitions can improve parallelism and consumer throughput.

3.1.2 Consumer Parallelism

- A single consumer instance reads from one or more partitions, but it can be overwhelmed if the data rate exceeds its capacity.
- Adding more consumers in a consumer group increases parallelism, as Kafka reassigns partitions to balance the load.

3.1.3 Fetch Configuration

- `fetch.min.bytes`: Minimum amount of data (in bytes) the broker returns for a fetch request. Larger values reduce fetch requests but may introduce latency.
- `fetch.max.bytes`: Maximum amount of data returned in a single fetch response. A higher value allows fetching larger batches of messages, improving throughput.
- `fetch.max.wait.ms`: Maximum time the broker waits before responding to a fetch request. A higher value can increase batch sizes and throughput but may increase latency.

For more details, see the Confluent Cloud Client Optimization Guide - Consumer Fetching.

3.1.4 Batch Size

- Consumers process messages in batches for better efficiency. Larger batches reduce processing overhead but require sufficient memory.
- Configuration: `max.poll.records` controls the number of records fetched in a single poll.

3.1.5 Message Size

- Larger messages can reduce throughput if the network or storage systems are bottlenecks. Use compression (e.g., `lz4`, `snappy`) to optimize data transfer.

3.1.6 Network Bandwidth

- Network speed between Kafka brokers and consumers is critical. A consumer running on a limited-bandwidth network will see reduced throughput.

3.1.7 Deserialization Overhead

- The time required to deserialize records impacts throughput. Efficient deserialization methods (e.g., Avro, Protobuf with optimized schemas) can help.

3.1.8 Broker Load

- Broker performance and replication overhead impact the throughput seen by consumers. If brokers are under heavy load, consumer throughput may decrease.

3.1.9 Consumer Poll Frequency

- Consumers must frequently call `poll()` to fetch messages. If the consumer spends too much time processing messages between polls, throughput can drop.

3.1.10 System Resources

- CPU, memory, and disk I/O on the consumer's machine affect how fast it can process data.

3.2 Typical Consumer Throughput

- **Single Partition Throughput**: A single consumer reading from a single partition can typically achieve **10-50 MB/s** or higher, depending on record size, compression, and hardware.
- **Multi-Partition Throughput**: For a consumer group reading from multiple partitions, throughput can scale linearly with the number of partitions (subject to other system limits).

3.3 Seven Strategies to Improve Consumer Throughput

1. **Increase Partitions**: Scale partitions to allow more parallelism.
2. **Add Consumers**: Add more consumers in the consumer group to distribute the load.
3. **Optimize Fetch Settings**: Tune `fetch.min.bytes`, `fetch.max.bytes`, and `fetch.max.wait.ms`.
4. **Batch Processing**: Use `max.poll.records` to fetch and process larger batches.
5. **Compression**: Enable compression to reduce the amount of data transferred.
6. **Efficient SerDe (Serialization/Deserialization)**: Use optimized serializers and deserializers.
7. **Horizontal Scaling**: Ensure consumers run on high-performance hardware with sufficient network bandwidth.

By optimizing these factors, Kafka consumers can achieve higher throughput tailored to the specific use case and infrastructure.

4.0 Resources

4.1 Optimization Guides

- Optimize Confluent Cloud Clients for Throughput
- Choose and Change the Partition Count in Kafka

4.2 Confluent Cloud Metrics API

- Confluent Cloud Metrics API
- Confluent Cloud Metrics API: Metrics Reference
- Confluent Cloud Metrics

4.3 Confluent Kafka Python Client

- Confluent Kafka Python Client Documentation