

Kafka Topics Partition Count Recommender Application

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 seven-day period, it identifies the peak 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**
 - **1.1 Download the Application**
 - **1.2 Configure the Application**
 - **1.2.1 Create the `.env` file**
 - **1.2.2 Using the AWS Secrets Manager (optional)**
 - **1.3 Run the Application**
 - **1.3.1 Did you notice we prefix `uv run` to `python src/app.py`?**
 - **1.3.2 Troubleshoot Connectivity Issues (if any)**
- **2.0 How the app calculates the recommended partition count**
- **3.0 Unlocking High-Performance Consumer Throughput**
 - **3.1 Key Factors Affecting Consumer Throughput**
 - **3.1.1 Partitions**
 - **3.1.2 Consumer Parallelism**
 - **3.1.3 Fetch Configuration**
 - **3.1.4 Batch Size**
 - **3.1.5 Message Size**
 - **3.1.6 Network Bandwidth**
 - **3.1.7 Deserialization Overhead**
 - **3.1.8 Broker Load**
 - **3.1.9 Consumer Poll Frequency**
 - **3.1.10 System Resources**
 - **3.2 Typical Consumer Throughput**
 - **3.3 Seven Strategies to Improve Consumer Throughput**
- **4.0 Resources**
 - **4.1 Optimization Guides**
 - **4.2 Confluent Cloud Metrics API**
 - **4.3 Confluent Kafka Python Client**

1.0 To get started

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
```

1.2 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:

```
BOOTSTRAP_SERVER_URI=<YOUR_BOOTSTRAP_SERVER_URI>
CONFLUENT_CLOUD_API_KEY=<YOUR_CONFLUENT_CLOUD_API_KEY>
CONFLUENT_CLOUD_API_SECRET=<YOUR_CONFLUENT_CLOUD_API_SECRET>
INCLUDE_INTERNAL_TOPICS=False
KAFKA_API_KEY=<YOUR_KAFKA_API_KEY>
KAFKA_API_SECRET=<YOUR_KAFKA_API_SECRET>
KAFKA_CLUSTER_ID=<YOUR_KAFKA_CLUSTER_ID>

REQUIRED_CONSUMPTION_THROUGHPUT_FACTOR=
<YOUR_REQUIRED_CONSUMPTION_THROUGHPUT_FACTOR>

USE_SAMPLE_RECORDS=<True|False>
SAMPLE_SIZE=1000

TOPIC_FILTER=

USE_AWS_SECRETS_MANAGER=<True|False>
AWS_REGION_NAME=<YOUR_AWS_REGION_NAME>
CONFLUENT_CLOUD_API_KEY_AWS_SECRETS=
<YOUR_CONFLUENT_CLOUD_API_KEY_AWS_SECRETS>
KAFKA_API_KEY_AWS_SECRETS=<YOUR_KAFKA_API_KEY_AWS_SECRETS>
```

The environment variables are defined as follows:

Environment Variable Name	Description
<code>BOOTSTRAP_SERVER_URI</code>	The bootstrap server URI for your Kafka cluster (e.g., <code>pkc-xxxxxx.us-east-1.amazonaws.com:9092</code>).
<code>CONFLUENT_CLOUD_API_KEY</code>	Your Confluent Cloud API Key.
<code>CONFLUENT_CLOUD_API_SECRET</code>	Your Confluent Cloud API Secret.
<code>INCLUDE_INTERNAL_TOPICS</code>	Set to <code>True</code> if you want to include internal topics in the analysis; otherwise, set to <code>False</code> .
<code>KAFKA_API_KEY</code>	Your Kafka API Key.
<code>KAFKA_API_SECRET</code>	Your Kafka API Secret.
<code>KAFKA_CLUSTER_ID</code>	The ID of your Kafka cluster (e.g., <code>lkc-123xyz</code>).
<code>REQUIRED_CONSUMPTION_THROUGHPUT_FACTOR</code>	A multiplier to scale the peak consumption for future demand forecasting (e.g., <code>1.5</code> for 150%).
<code>USE_SAMPLE_RECORDS</code>	Set to <code>True</code> if you want to sample records for analysis; otherwise, set to <code>False</code> .
<code>SAMPLE_SIZE</code>	The number of records to sample if <code>USE_SAMPLE_RECORDS</code> is set to <code>True</code> (e.g., <code>1000</code>).
<code>TOPIC_FILTER</code>	A comma-separated list of topic names to include in the analysis. Leave empty to include all topics.
<code>USE_AWS_SECRETS_MANAGER</code>	Set to <code>True</code> if you want to use AWS Secrets Manager to manage your secrets; otherwise, set to <code>False</code> .
<code>AWS_REGION_NAME</code>	The AWS region where your secrets are stored (e.g., <code>us-east-1</code>).
<code>CONFLUENT_CLOUD_API_KEY_AWS_SECRETS</code>	The name of the AWS Secrets Manager secrets that contains your Confluent Cloud API Key and secret.
<code>KAFKA_API_KEY_AWS_SECRETS</code>	The name of the AWS Secrets Manager secrets that contains your Kafka Cluster API Key, API Secret, Kafka Cluster ID, and bootstrap server URI.

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

1.3 Run the Application

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

Then enter the following command below to run the application:

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`?

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

- 4 / 8

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.

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:

- $1.22\text{GB/s} = 1250\text{MB/s}$

2. Divide the target throughput by the consumer's capacity:

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

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

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

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.

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](#)