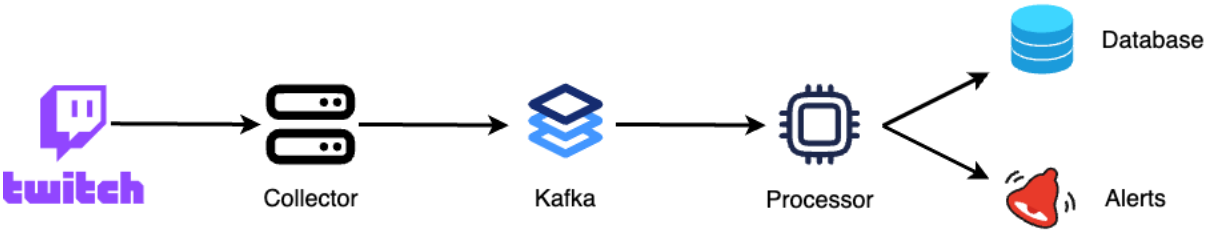
Twitch API Data Collection System

Architectural Overview

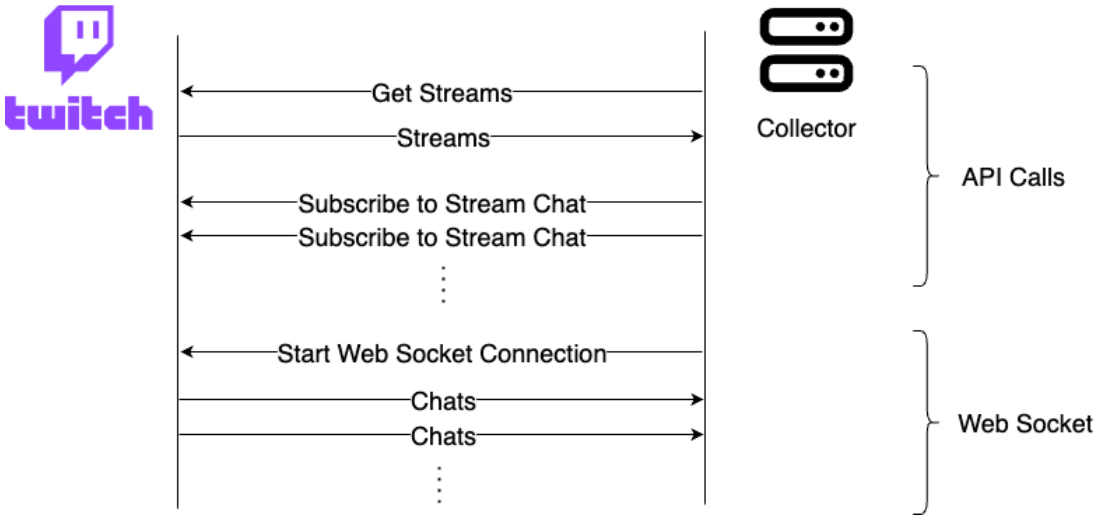


For a resilient and reliable real-time twitch data collection system, I propose the system with the high-level architecture above. Below I break down the components and give details about each service. The design involves collectors which fetch the data from the twitch api, saving the videos to file storage and then passing the chat messages to a message broker to which a processing service subscribes to process the messages and save them to the database while monitoring and alerting the user(s) when the messages received surpass a user-set threshold.

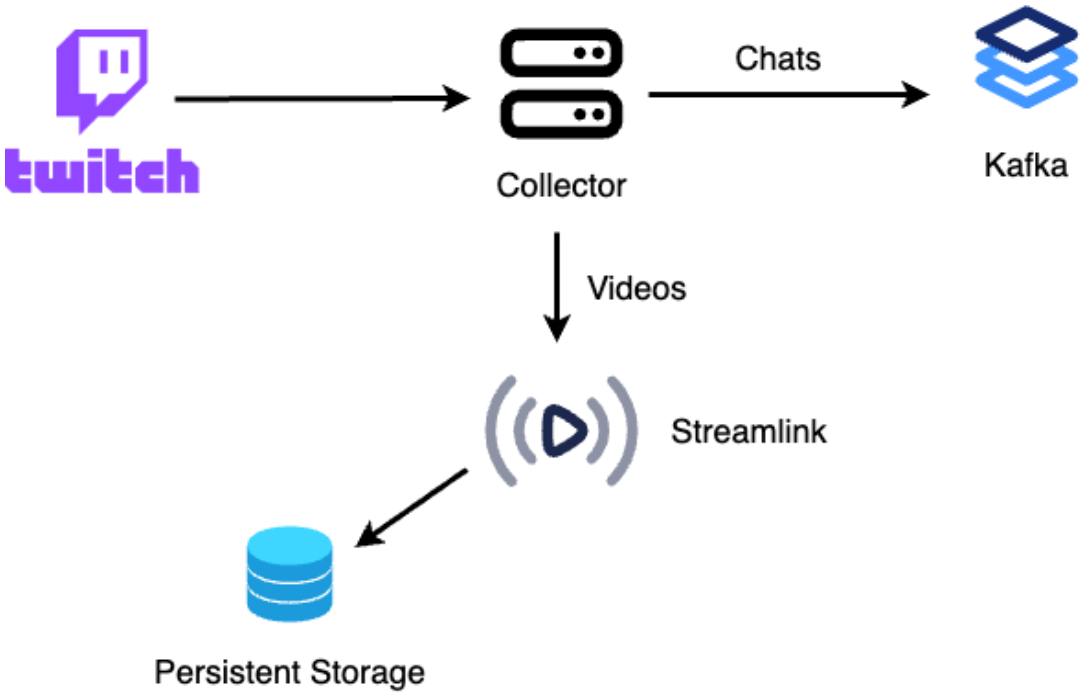
Deep Dive

- **Twitch API**
  - This is where we shall be fetching the video and chat data from.
  - They implement rate limits on their endpoints, we shall look at how our system will work with them for effective data collection.
  - As with any other system, we have to assume that in some cases, the requests to this API may fail for some reason like network failures, timeouts, etc. Our data collection system will have mechanisms in place to work around these like retries, reconnections, etc.
- **Data Collection Service**
  - This is the service that calls the twitch API for data, it subscribes to a twitch chat or stream and then connects to the twitch web socket

service to receive chat messages from these streams.



- This service will save the videos collected to persistent file storage volume. Using `streamlink` , this service can initiate a download of the stream to a file for each stream.



- Ideally the service only needs to access the twitch address and the code repository, for enhanced security, we can put a firewall around it and accept outbound traffic to only the message queue and inbound traffic from only twitch and the code repository.

    - For redundancy and more throughput we can add multiple instances of this service, we do have to synchronise them in terms of usage of the limit, we have to avoid hitting the limit and having to at best authorise again or at worst, wait for some 30 minutes to use the api with the twitch server again.

- **Message Queue**
    - To smoothly process and store the data from twitch, we add a message queue to enable asynchronous relay of data from the collectors to the processing service (the next component).
    - This allows the processing service to get the messages once it is ready to process them and not as they come in, this makes the system more stable and reliable, the processing service gets overwhelmed even if there are spikes in data received from Twitch.
    - *Apache Kafka* is a great option for this use case as it handles high throughput well with low latency.
- **Processing Service**
    - Once the data is collected, it needs to be stored to the database and this service is the one that will do that. It will fetch the data from the queue and then save it to the database
    - In order not to overwhelm this service with a lot of data, we save the videos at the point of collection since no processing will be done on them. We save the videos as they come in. We can use the `stream_id` as the name of the file and this will save us the synchronisation between the processing service and the data collection service.
    - We could do extra manipulation on the chat messages such as sentiment analysis, translate to or from different languages, add extra analytics of the chats before saving the chat data to the database
    - This service will also be the one to count the chat messages and send alerts once a certain threshold set by the user is reached.
    - To scale the system, multiple instances can also be added to crunch down more data and save it to the database.
- **Database**
    - The NoSQL database, MongoDB, will be used to store the chat messages for its speed, reliability, and scalability.

    - We can easily scale key-value stores with horizontal scaling by sharding the database using a hashing algorithm over the keys and store the data in different instances of the database.

- **Alerts**
    - We want to be able to receive real-time alerts from the system once a threshold of chat messages is reached.
    - The simple notification service will be used, we can breakdown the messages into different topics for different kinds of chats (depending on the theme of the twitch broadcasters or topics), the users can receive the messages as emails or even SMS (Amazon SNS covers a wide range of countries for SMS)
    - Amazon SNS also encrypts the messages so they will securely reach the receiver
    - In case of failure, Amazon SNS also has a dead letter queue that we can use to resend the messages so that they reliably reach the users.
- **Monitoring**
    - To keep the system reliably running, we need to monitor the different components and we can use `Prometheus` for this.
    - We set up alerts from Prometheus to developers, system administrators, and other stakeholders in case service instances go down for some reason.

Conclusion

Designing and implementing a reliable, scalable, and efficient twitch data collection system is not an easy task. There are many factors to put into consideration and the design does not reflect what will happen in a real production environment but it serves as a plan and a guide into the implementation of such said system. Some of the critical challenges and design decisions include the ones below:

- Rate limiting from twitch which could mean being locked out of their servers for a certain period of time, I had to make sure this does not happen by making the collectors wait in case the rate limit was about to get hit until it was okay to fetch for data again

- Choosing a message broker, the main contenders were Apache Kafka and RabbitMQ. Kafka was chosen for its high throughput, simplicity and low latency
- Deciding how to record the live stream, the best way to do this was to use the `streamlink` tool which uses `ffmpeg` under the hood. This tool then saves the stream as it comes in and saves it to persistent storage. The beauty with this approach is that because these operations are mostly IO bound operations, we can allocate threads for each stream multiple times and we can spawn up as many threads as 10,000 or more depending on the server operating system.
- Picking between web-hooks or web sockets for receiving chat messages from twitch because twitch accepts both. Web sockets are faster and a relatively recent protocol compared to web-hooks therefore they are the natural choice, the challenge with this is that the system needs to reconnect to the twitch web socket server every time the system joins a twitch chat or leaves one.