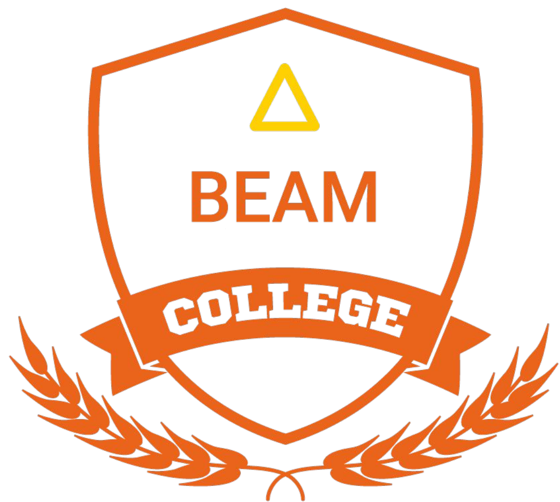


Real-Time Stateful Processing & Anomaly Detection Of Video Data

Aditya Shukla
Darshan Kanade

Agenda

- Video processing challenges
- Workflow/System overview
- Data Ingestion
- Anomaly Detection Pipeline
- What is Stateful Processing?
- Demo Walkthrough

Video Processing Challenges

Massive Volume

01

- **A single 1080p camera** at 30 frames per second generates roughly **3 gigabytes of video per hour**.

- If you have **50 cameras**, that's **~150 gigabytes an hour**.

- You cannot store all of that and batch process it later — by the time your job runs, the event is long over.

Always-on latency

02

- If someone breaks in at 2 am, you don't want to be notified at 5 am when the job finishes.

- Video processing requires low latency and real-time solution.

Noise

03

- Video feeds are full of things that look like motion **but aren't intrusions**, like:
  - An insect crossing the lens.
  - A light flickering.
  - A cloud moving past a window and changing the brightness of a frame.

- If you flag every anomalous frame, it results in a lot of false alarms.

Context across frames

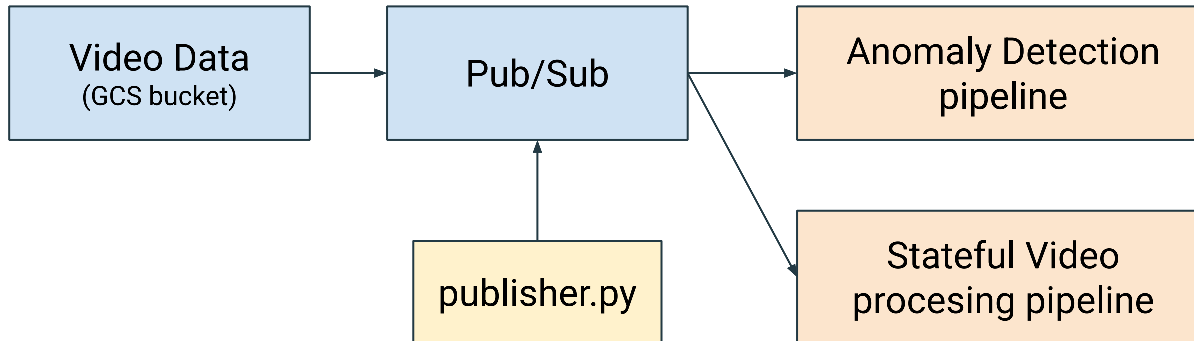
04

- Video stream analysis works with frame extraction.

- Anomalies don't exist in isolation - we need to capture temporal context to predict anomalies meaningfully.

Workflow setup for the Demo

1. Burglary video stored in GCS bucket for demo
2. publisher.py file simulates the streaming by accessing video from GCS and pushing frames to pub/sub topic.
3. The pipeline then branches into 2 step workflow -
    a. Anomaly Detection using Windowing & RunInference
    b. Stateful video processing pipeline



Pub/Sub & Publisher walkthrough

1. Download the video from **GCS to a local temp file**
2. Read frames using **OpenCV**
3. **Frame Skip**: **cost-versus-resolution tradeoff**

   We don't publish every frame - Publishing every frame at full resolution would flood Pub/Sub and make inference expensive. For intrusion detection, 5fps is more than enough — a person walking through frame will still appear in multiple consecutive published frames

1. For each frame we keep, we **JPEG-encode** it - Base64 encoding is necessary because Pub/Sub expects data in bytes.
2. Package the message_data and publishes to pub/sub.

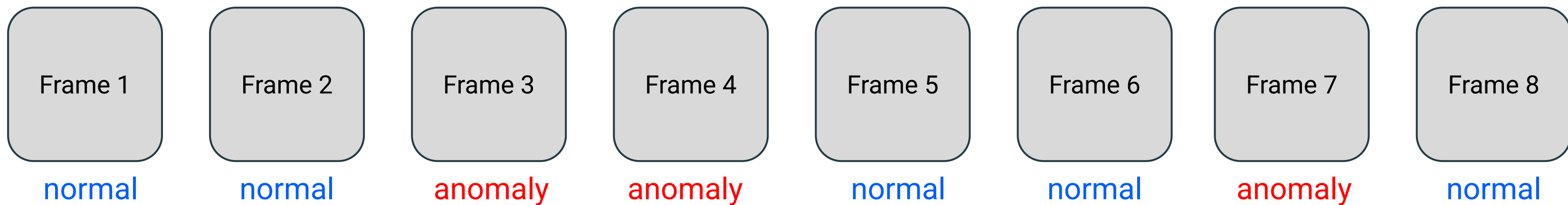
```
cap = cv2.VideoCapture(local_path)
fps = cap.get(cv2.CAP_PROP_FPS) or 25

if frame_idx % FRAME_SKIP == 0:
```

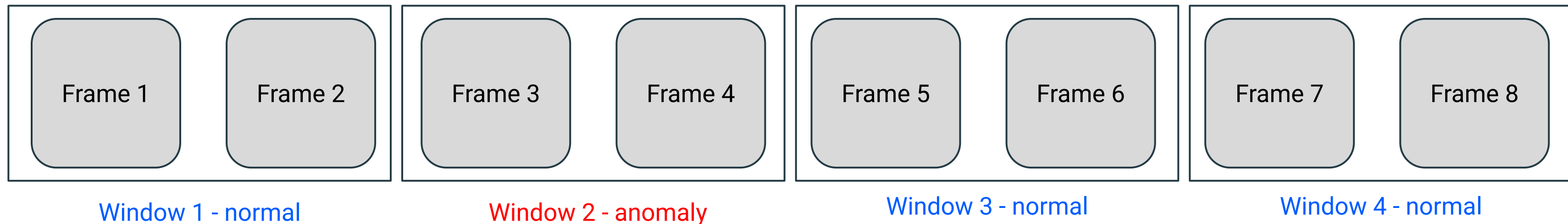
```
message_data = json.dumps({
            "frame_idx":    frame_idx,
            "timestamp_ms": timestamp_ms,
            "width":        frame.shape[1],
            "height":       frame.shape[0],
            "jpeg_b64":     frame_bytes,
        }).encode("utf-8")
```

Conceptual Flow

1. Two steps to anomaly detection:
    a. Per-frame detection



    b. Aggregated, contextual detection (using windowing)



Anomaly Detection Walkthrough

- We use both the methods
  - First perform **anomaly scoring per-frame**
  - Then take **mean** over the defined **fixed window**
- **Why?** In use cases like intrusion detection, anomalies are context-driven. We need temporal validation
  - We dont want to capture anomalies caused by random birds or insects in frame
  - We take a mean of 10 second window and **only if 25%+ frames were anomalous,** we tag it as an anomaly.

Anomaly Detection Pipeline: 1

```python
model_handler = KeyedModelHandler(
    TFSavedModelHandler(model_gcs_path=known_args.model_gcs_path)
)

with beam.Pipeline(options=pipeline_options) as p:

    # ── 1. Ingest + preprocess ──────────────────────
    keyed_frames = (
        p
        | "ReadFromPubSub"      >> ReadFromPubSub(
            subscription=known_args.subscription,
            with_attributes=False,
        )
        | "DecodeAndPreprocess" >> beam.ParDo(DecodeAndPreprocessFn())
        | "FixedWindow"         >> beam.WindowInto(
            window.FixedWindows(known_args.window_size),
            trigger=beam.trigger.AfterWatermark(
                early=beam.trigger.Repeatedly(
                    beam.trigger.AfterProcessingTime(5)
                )
            ),
            accumulation_mode=beam.trigger.AccumulationMode.ACCUMULATING,
            allowed_lateness=0,
        )
    )
```

1. Decode Pub/Sub message, resize and normalize the data to match model expectation - and **yield (key, arr)**
2. Each keyed frame **(key, arr)** is assigned to a 10-second window
3. **Beam.trigger** fires every **5 seconds** to get partial results and fires finally when the watermark passes.

Anomaly Detection Pipeline: 2

```python
# —— 2. Inference — one call per frame, no duplication ——————————
frame_scores = (
    keyed_frames
    | "RunInference"       >> RunInference(model_handler)
    # Compute per-frame MSE and attach dummy key so GBK groups by wi
    | "ExtractFrameScore" >> beam.Map(
        lambda kv: (
            0,
            {
                "frame_idx":    json.loads(kv[0])["frame_idx"],
                "timestamp_ms": json.loads(kv[0])["timestamp_ms"],
                "recon_error":  float(np.mean(
                    (kv[1].example - kv[1].inference) ** 2
                )),
            }
        )
    )
)
```

1. **TFSavedModelHandler.load_model() runs once per worker** — it downloads the SavedModel from GCS to a temp directory and loads it with **tf.saved_model.load()**.
2. We calculate per-frame anomaly score:
   a. Notice we also **pass 0 - a dummy key** - so we can use **beam.GroupByKey()** over the window.

Anomaly Detection Pipeline: 3

```
# —— 3. Group all frame scores in each window, then score the window ——
scored = (
    frame_scores
    | "GroupByWindow"   >> beam.GroupByKey()
    | "ScoreWindow"     >> beam.ParDo(
        WindowAnomalyScoringFn(
            threshold=known_args.anomaly_threshold,
            min_anomaly_fraction=known_args.min_anomaly_fraction,
        )
    ).with_outputs(WindowAnomalyScoringFn.ANOMALY_TAG, main="all_windows")
)
```

1. Now we have one element per window containing every frame's reconstruction error.
2. We use the dummy key from previous step to aggregate all frames scores in each window.
3. If the window_anomaly_score > min_anomaly_fraction, we flag the window as an anomaly.

Stateful Processing Pipeline Use Case

Goal: To generate sessions of activity in a surveillance footage in case of any intrusion

- Maximum length of Session should be 5 minutes
- Don’t end the session immediately when no motion is detected as the intruder might be hidden behind something or might have left the frame briefly
- In such cases wait for 30seconds before ending the session
- Session details can include
  - start and end time of session
  - duration of session
  - Number of frames processed
  - Frames themselves (if needed), etc.

Stateful Processing - State & Timers

State lets you remember things

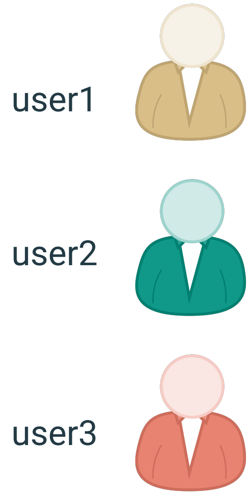
State = memory

Timers let you decide when to act on what you remembered

Examples:

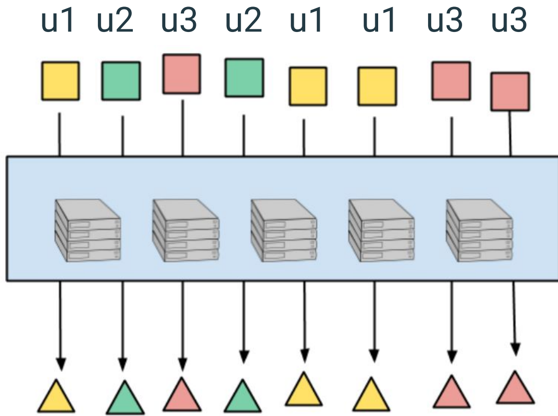
- Do something after every 5 minutes
- Raise an alert if condition persists for 30 seconds

Examples of Illegal Parking Detection



8:00 9:00 10:00 11:00 12:00 13:00 14:00 15:00 16:00 17:00 18:00 19:00 20:00 21:00

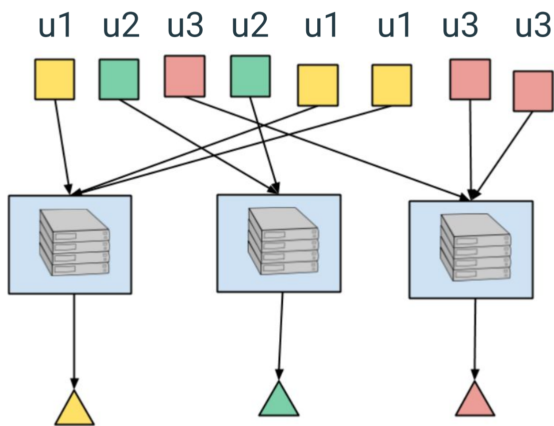
Element wise Processing
(ParDo, Map, FlatMap, etc.)



Limitations:

- Does not provide shared context
- Cannot remember things

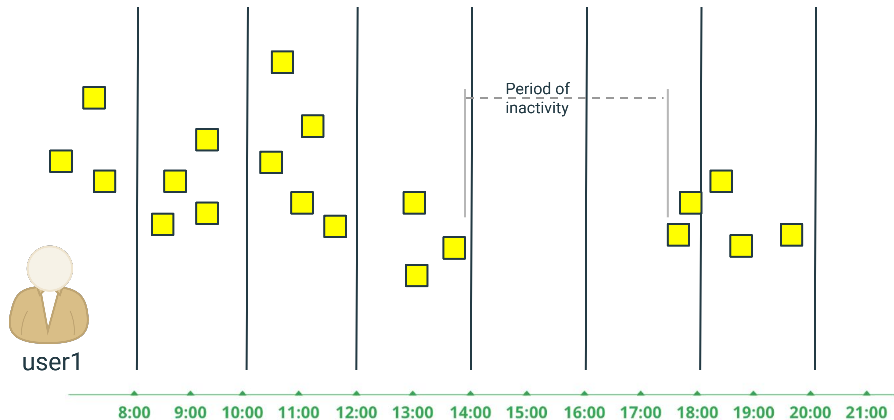
Per-Key aggregation (GroupByKey, CombinePerKey, etc.)



Limitations:

- It ignores time
- Does not let you decide when to emit results

  Eg. Give me count of elements every 5 minutes



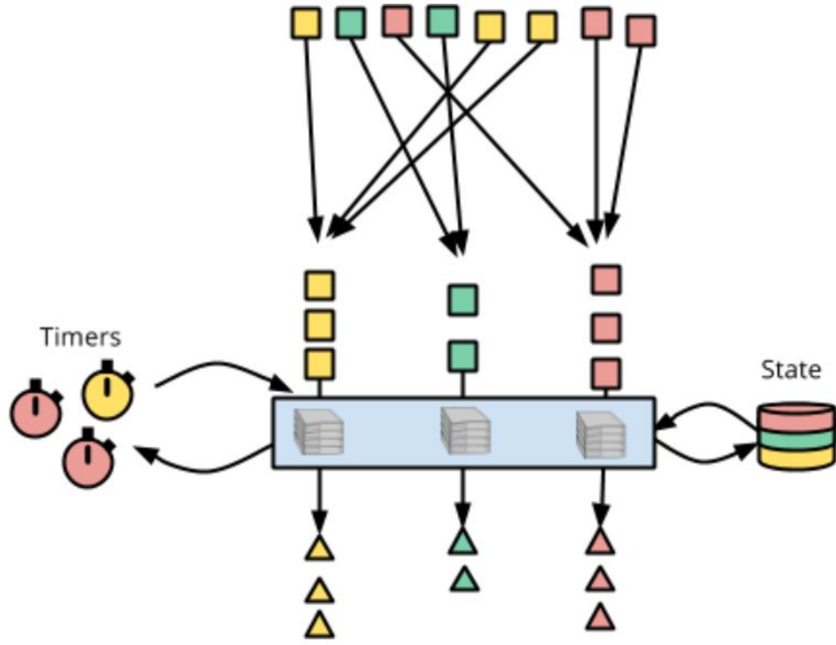
Limitations:

- Does not provide more control
- Cannot do custom logic

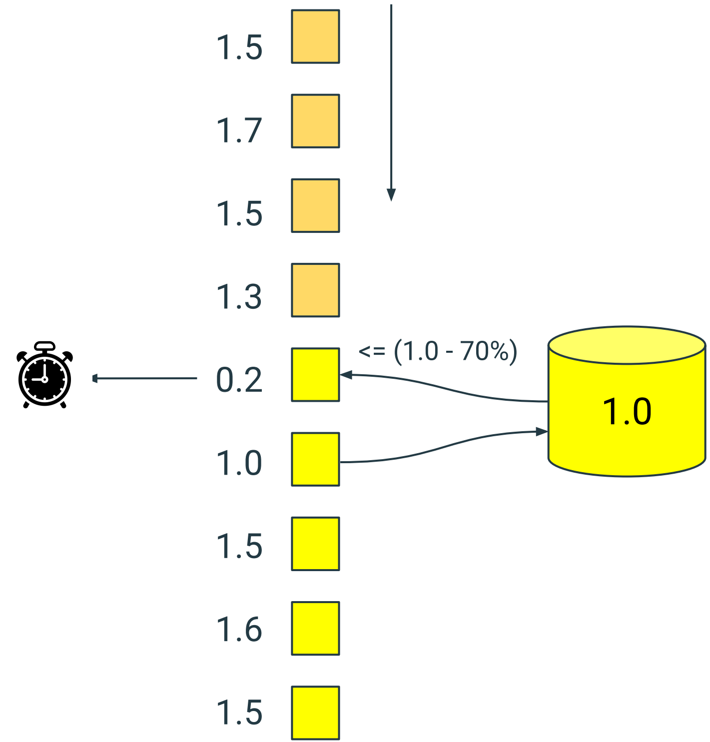
Examples:

- Give me duration of inactivity
- Alert when a value changes from a to b
- Continuously update counts but emit only every 20 seconds

State and Timers



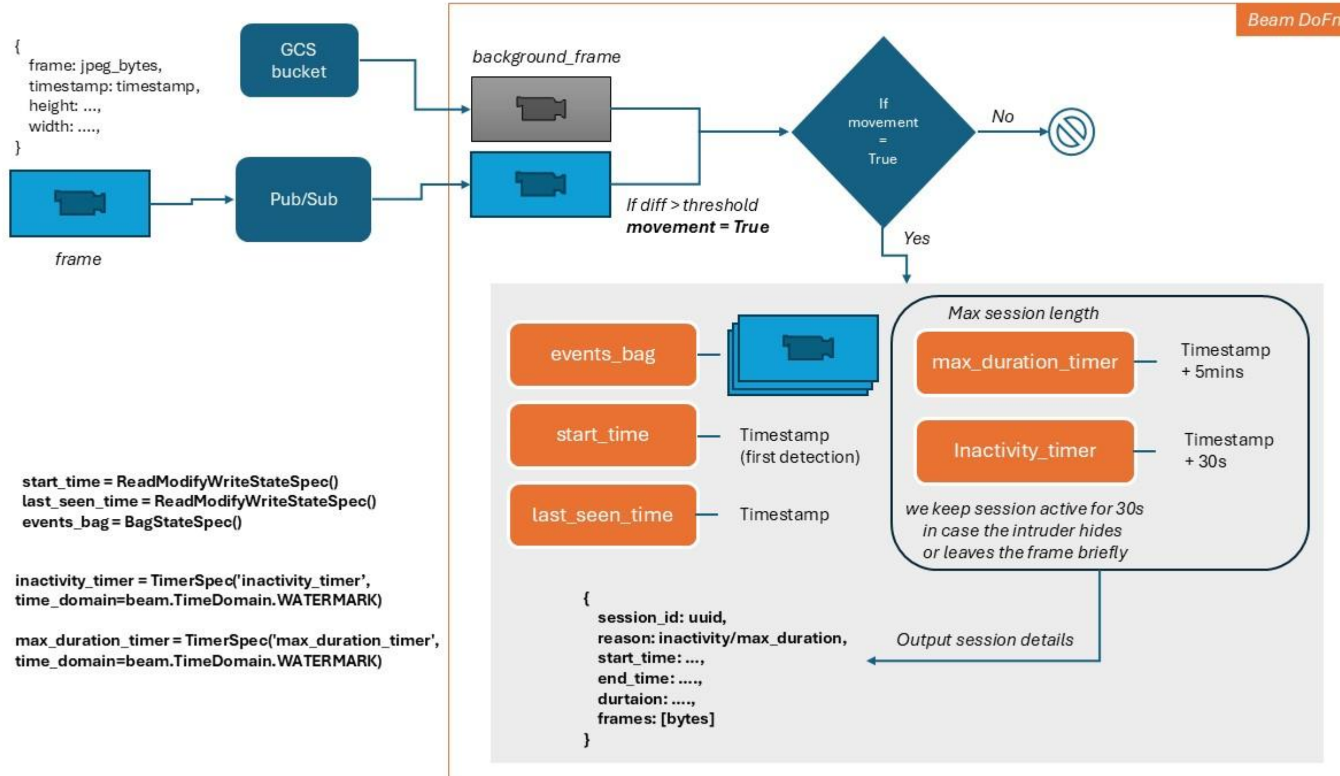
State is per Key per Window



Types of Timers

- Event Time - *"when the event happened"*
  - Triggered based on the **timestamp of the data (event time)**
  - Aligned with **when the event actually occurred**, not when it was processed
  - Works with **watermarks** to handle late or out-of-order data
- Processing Time - *"when the system sees it"*
  - Triggered based on **system clock (wall-clock time)**
  - Independent of event timestamps

Workflow Diagram



Pipeline Walkthrough

```
options = PipelineOptions(**pipeline_options)
with beam.Pipeline(options=options) as p:
    frames = (
        p
        | "ReadFromPubSub"      >> ReadFromPubSub(
                topic = f'projects/{PROJECT_ID}/topics/{TOPIC_ID}',
                with_attributes=False,
            )
        | "Decode" >> beam.ParDo(DecodeMessage())
        | "AddTimestamps" >> beam.Map(lambda ele: window.TimestampedValue(ele, ele['timestamp_ms']/1000))
        | "Keys" >> beam.WithKeys(lambda ele: 'video')
        | "Session" >> beam.ParDo(DetectMovement())
        | "LogElements2" >> beam.LogElements()#with_window=True)
        )
```

Dataflow Pipeline Results

Dataflow / Jobs / Dataflow job details

- Overview
- Monitoring
- Jobs
- Pipelines
- Workbench
- Snapshots

Release Notes

intrusion-detection-pipeline   Stop   Create Snapshot   Archive   Import as pipeline   Share   Trigger   Send feedback

Job Graph | Execution Details | Job Metrics | Cost | Recommendations | Autoscaling

Job steps view: Graph view   Clear selection

ReadFromPubSub
Running
Data Lag: 37.35 sec
26% Total Walltime
Max Op Latency: < 1 sec
1 stage

Decode
Running
Data Lag: 37.35 sec

Logs   Hide   3   7

Job Logs | Worker Logs | Diagnostics | Data Sampling

Severity: Info   Filter   session_id   Search all fields and values   Max Time

Severity	Time	Summary

Streaming has paused   Restart streaming

Showing logs for time specified in query. To view more results update your query.

```
2026-04-23 02:11:59.282 IST  End session since max duration of 5 minutes reached. Here are the details: {'session_id': UUID('33286112-4395-4be4-9140-42e77c03f7c1'), 'reason': 'Max Duration', 'start_time': 1776890158.0, 'end_time': 1…
2026-04-23 02:12:18.005 IST  End session since max duration of 5 minutes reached. Here are the details: {'session_id': UUID('cdc43675-8889-4de5-94ab-94281ae71d9c'), 'reason': 'Max Duration', 'start_time': 1776890304.0, 'end_time': 1…
2026-04-23 02:14:14.219 IST  End session since max duration of 5 minutes reached. Here are the details: {'session_id': UUID('9047f215-f00d-4832-8032-88d78f6a2b19'), 'reason': 'Max Duration', 'start_time': 1776890521.0, 'end_time': 1…
2026-04-23 02:16:28.099 IST  End session since no movement detected for 30 seconds. Here are the details: {'session_id': UUID('5b03b986-ce0f-4aa2-94cf-68cb5c29804e'), 'reason': 'Inactivity', 'start_time': 1776890655.0, 'end_time': 1…
```

Showing logs for time specified in query. To view more results update your query.

```
2026-04-23 02:11:59.282 IST   End session since max duration of 5 minutes reached. Here are the details: {'session_id': UUID('33286112-4395-4be4-9140-
                              42e77c03f7c1'), 'reason': 'Max Duration', 'start_time': 1776890158.0, 'end_time': 1776890292.0, 'duration': 134.0,
                              'events_processed': 1091}
```

Thank you!