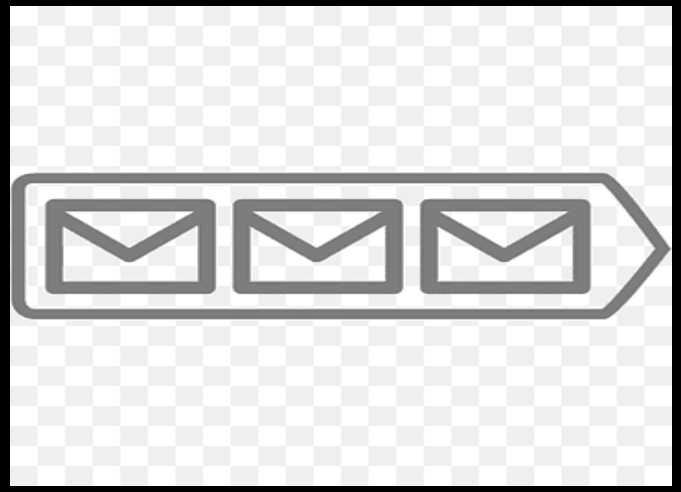
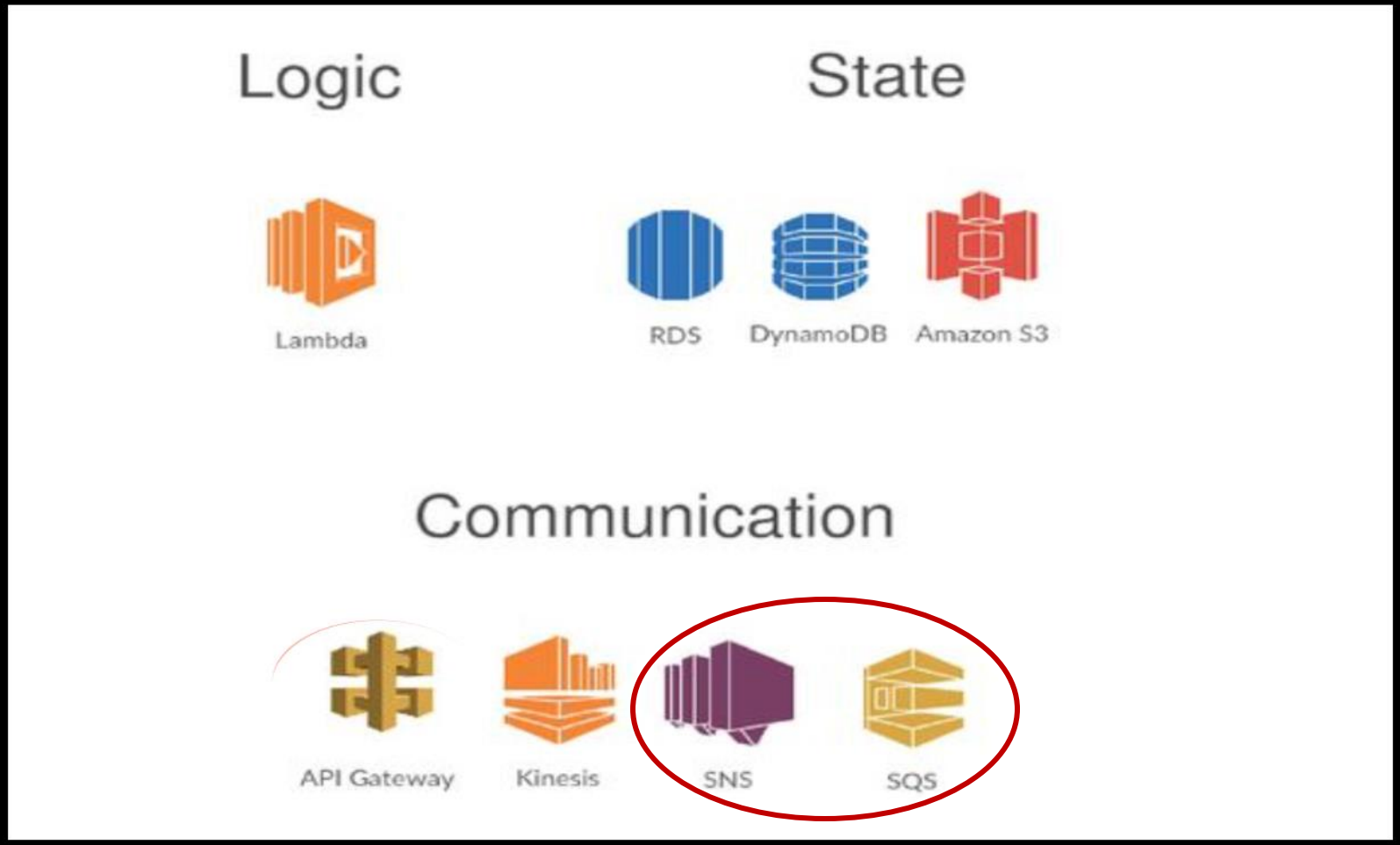


AWS Integration and Messaging Services.

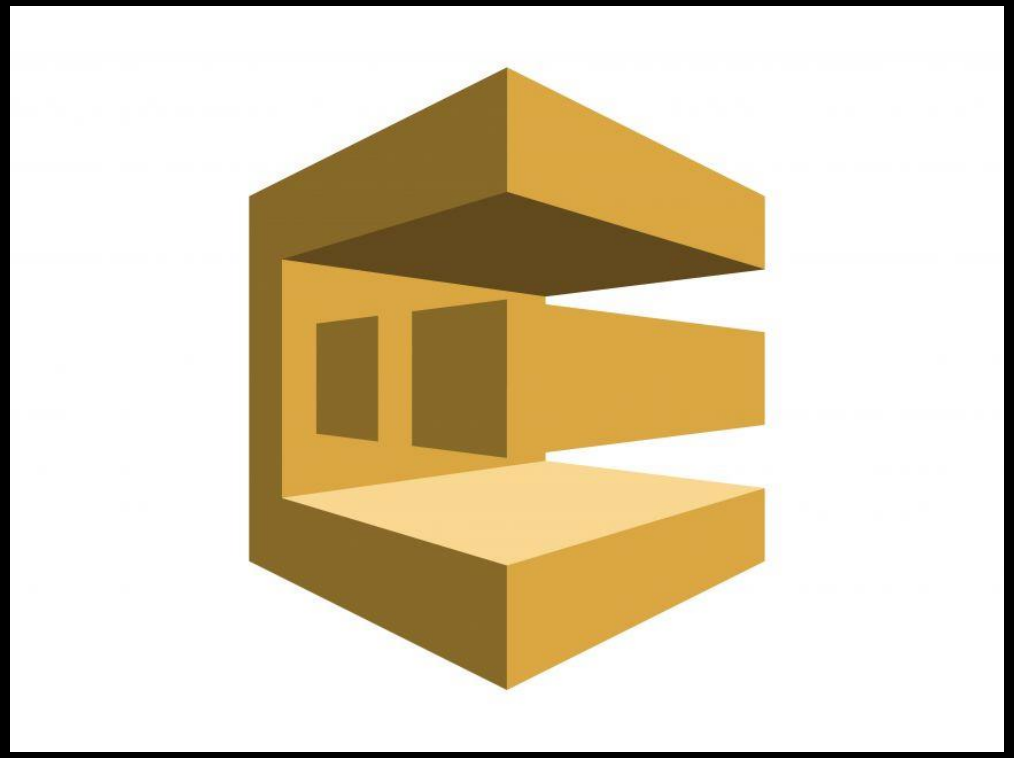
Components of a Serverless, Message-Driven application

(aka Event Driven Architecture - EDA)



Why do we need Messaging Services?

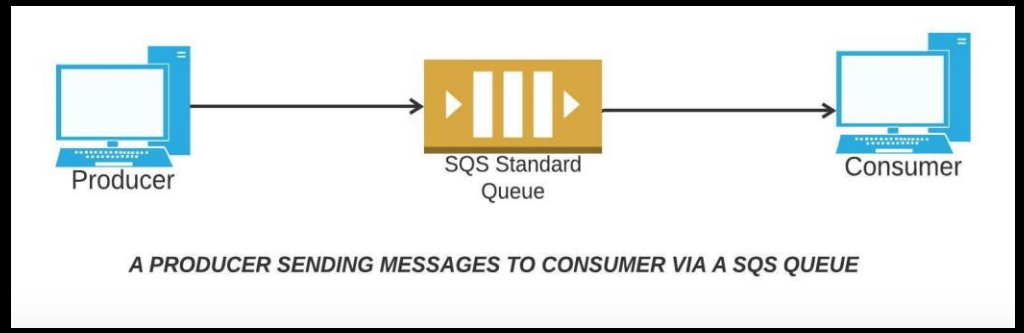
- Synchronous communication between compute components (Lambdas, EC2 instance) can be problematic if there are sudden spikes in demand or gaps in availability.
  - 1000 parallel requests to encode video uploads when usually the workload is a much smaller scale (10s).
- It's better to decouple compute components by using a range of AWS services/techniques:
  - SQS: queueing model.
  - SNS: pub/sub/subscribe model.
  - Data streams.
- These techniques result in:
  - Reduces latency; Increase availability; Reduces complexity (by decreasing dependency).

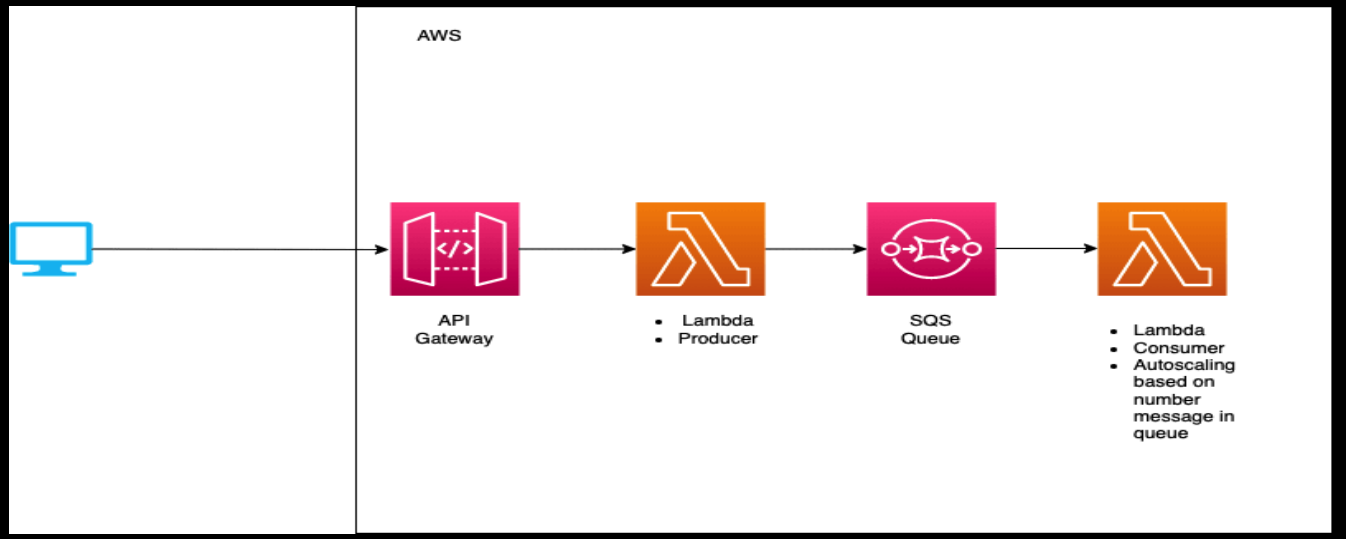


Simple Queue Service (SQS)

AWS SQS - Overview

- Compute component actors - Producers and Consumers.





SQS - Overview

- Oldest AWS offering (2006).
- Fully managed, distributed queueing service, used to decouple applications/components.
- Attributes:
  - Scalable - Unlimited throughput, unlimited number of messages in a queue.
  - Retention of messages: 4 days (default), maximum of 14 days.
  - Low latency (<10 ms on publish and receive).
  - Limitation of 256KB per message.
- Caveats:
  - Duplicate messages may occur, occasionally.
    - Consumer processing must be idempotent.
  - Can have out of order messages (best effort ordering).

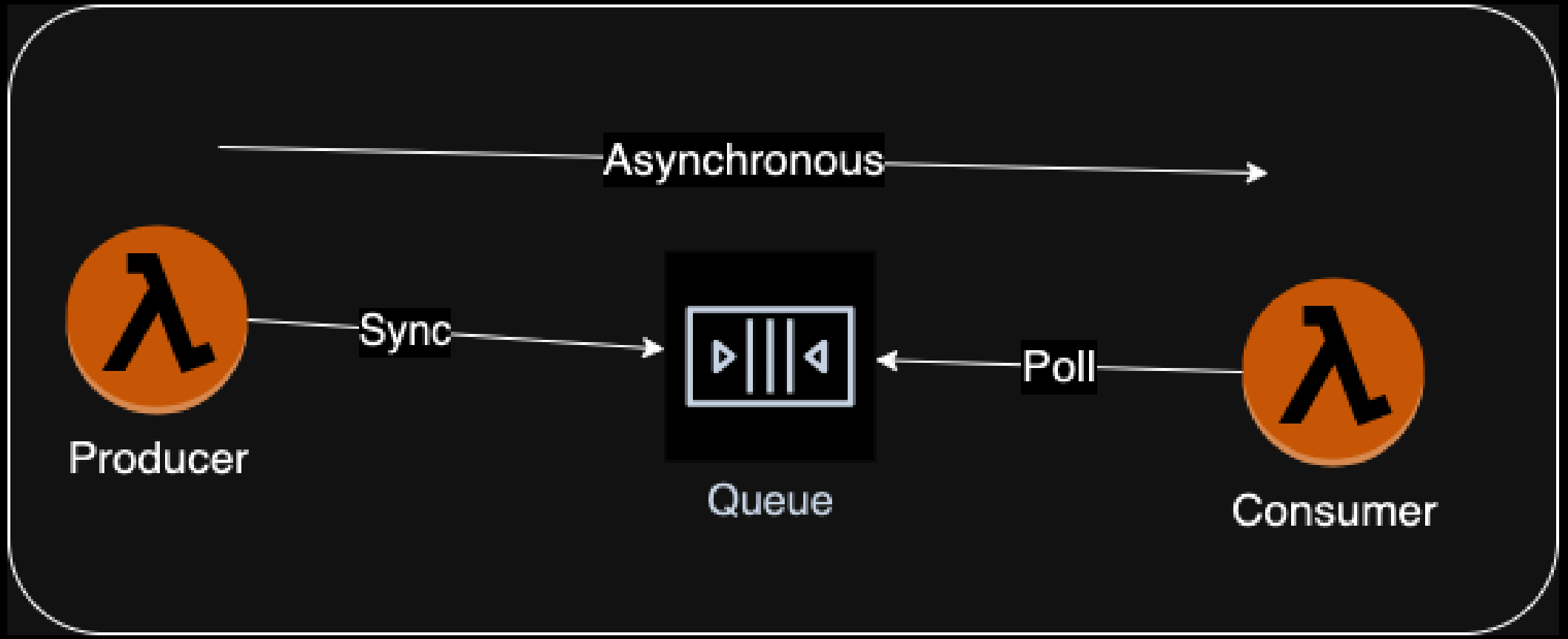
Basic Operations.

- Producer:
  - Publish/Write message to queue using SQS SDK.
    - SQS persists a message until a consumer deletes it (or its TTL expires – default 4 days).
    - e.g. Send an order to be processed

      Message = Order id + Customer id + Order details

- Consumer:
  1. Polls SQS for messages
  2. Receives batch response (<= 10 messages).
  3. Process the messages, e.g. validate & insert order into a d/b
  4. Delete the messages using the SDK.

Communication styles.



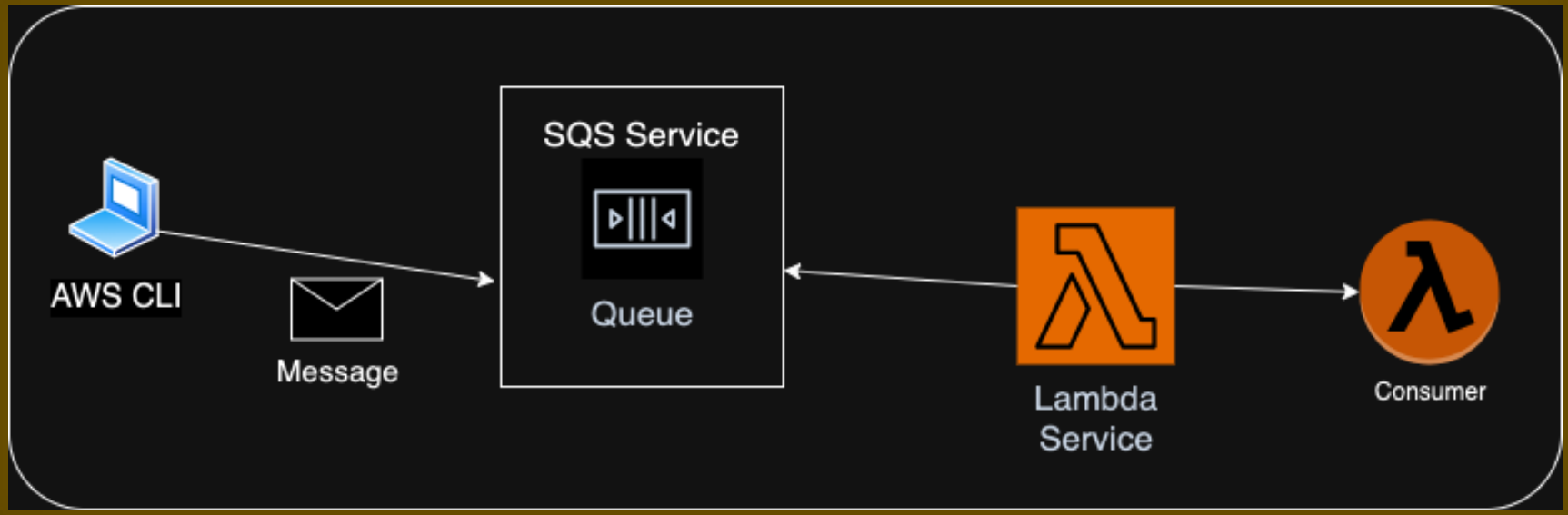
Security

- Encryption.
  - In-flight encryption using HTTPS API.
  - At-rest encryption using KMS keys.

- Access Controls: IAM policies to regulate access to the SQS API.

- SQS Access Policies (similar to S3 bucket policies).
  - Useful for cross-account access to SQS queues.
  - Useful for allowing other services (SNS, S3...) to write to an SQS queue.

Demo.



- The Lambda service polls the SQS for messages and calls the lambda function synchronously with a batch.
- If the function processes the batch without a failure/exception, the Lambda service deletes the batch from the queue.
  - Otherwise, the entire batch is ‘re-enabled’ in the queue (for reprocessing by the function/consumer).

Demo (CDK Infrastructure)

```
254
255   const demoQueue = new Queue(this, "Demo Queue");
256
257   const qConsumerFn = new NodejsFunction(this, "SQSConsumerFn", {
258     architecture: Architecture.ARM_64,
259     runtime: Runtime.NODEJS_16_X,
260     entry: `${__dirname}/../lambdas/consumeQMessages.ts`,
261     timeout: Duration.seconds(10),
262     memorySize: 128,
263   });
264
265   const eventSource = new SqsEventSource(demoQueue);
266   qConsumerFn.addEventSource(eventSource)
267
268   new CfnOutput(this, "Queue Url", { value: demoQueue.queueUrl });
269
```

- Recall, lambda function are triggered by an event.
- Here, the event source is a batch of messages from a queue (that's polled by the Lambda service).

Demo (Producer & Consumer)

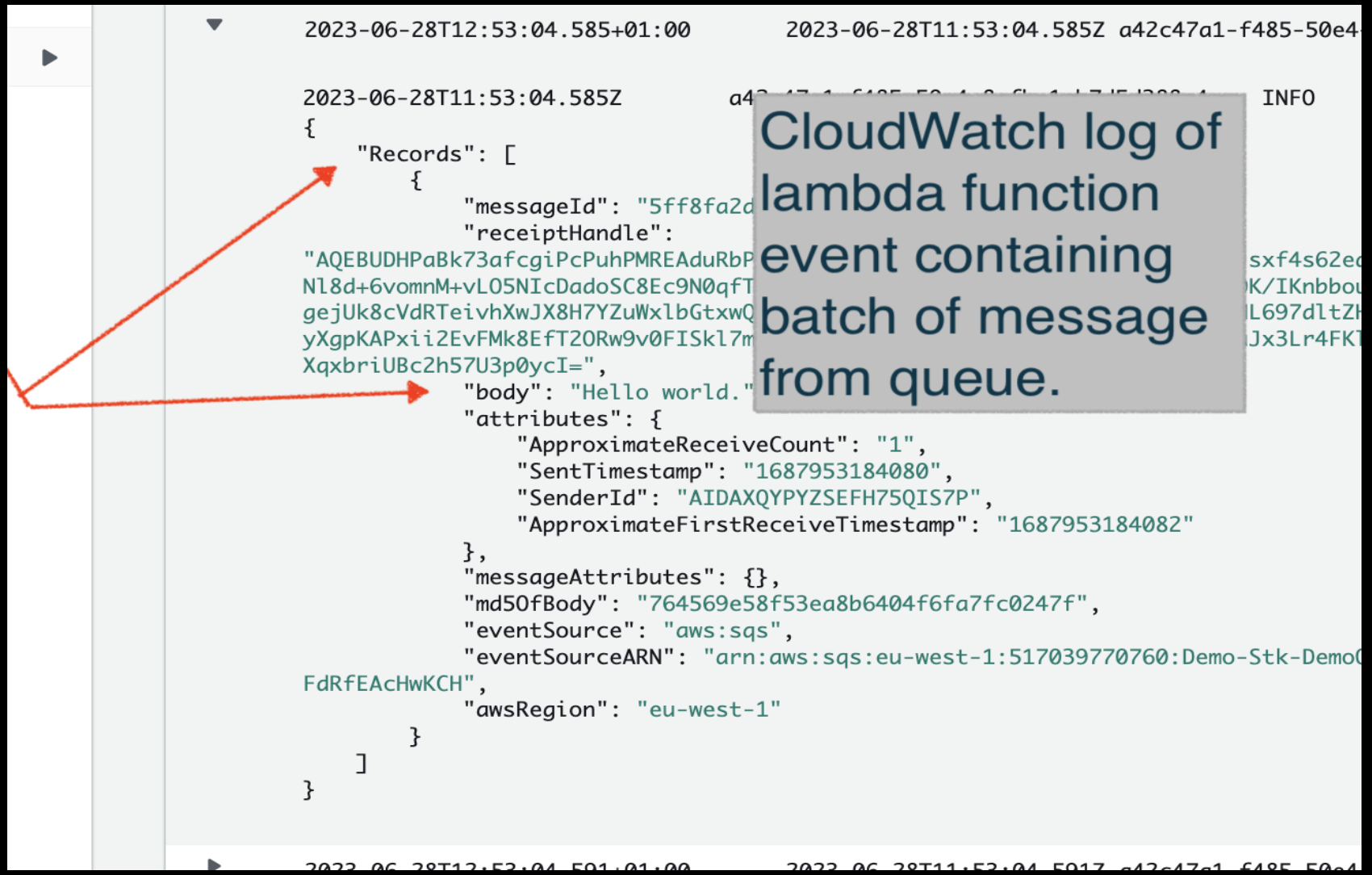
```
274
275  import { SQSHandler } from "aws-lambda";      You, 1 second ago • Ur
276
277  export const handler: SQSHandler = async (event) => {
278    try {
279      console.log("Event: ", JSON.stringify(event));
280      for (const record of event.Records) {
281        console.log("Message:  ", record.body);
282      }
283    } catch (error) {
284      console.log(JSON.stringify(error));
285    }
286  };
287
```

Batch

AWS CLI

```
$ aws sqs send-message
  --queue-url https://sqs.eu-west-1.amazonaws.com/517039770760/
  Demo-Stk-DemoQueueA7C0530A-FdRfEAcHwKCH
  --message-body "Hello world."
```

Demo (Handler event structure)



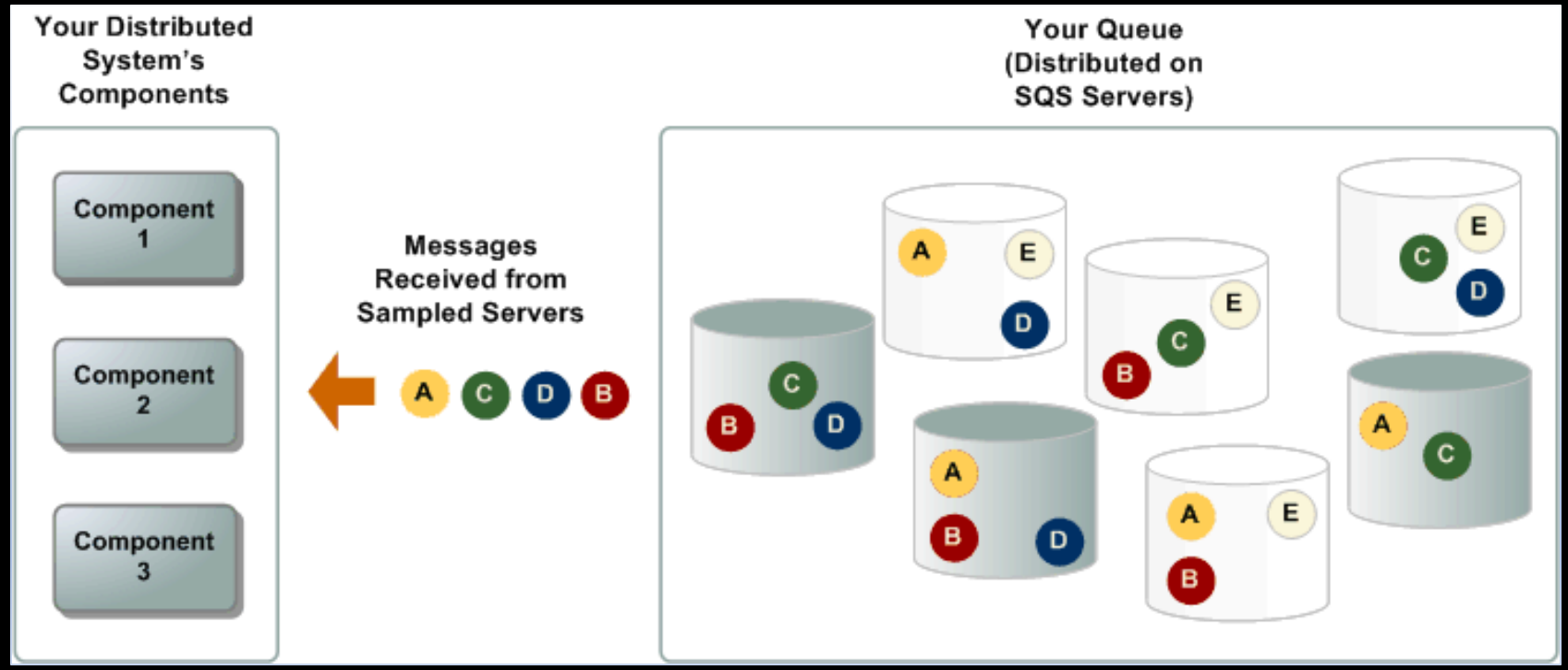
Demo (JSON messages)

- SQS serialize JSON messages → Handler must parse before processing.

Demo (JSON messages)

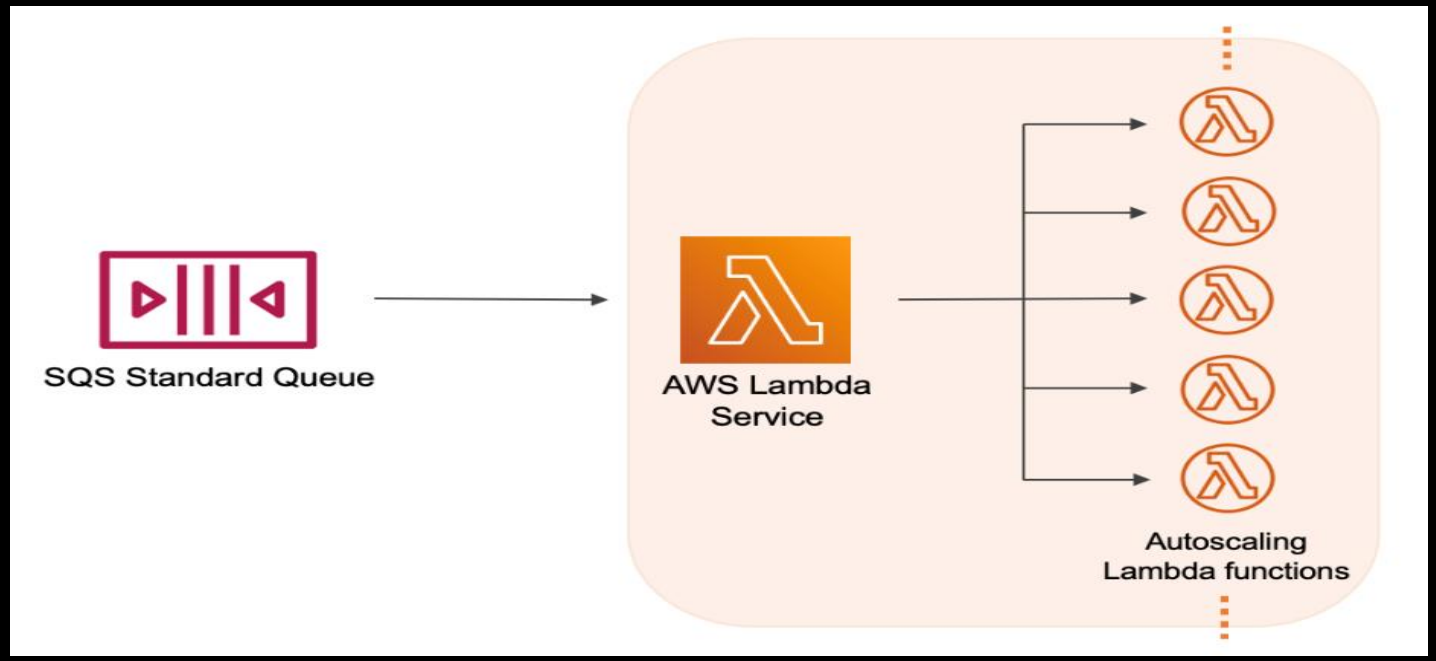
- The lambda handler (Consumer)

SQS is Resilient



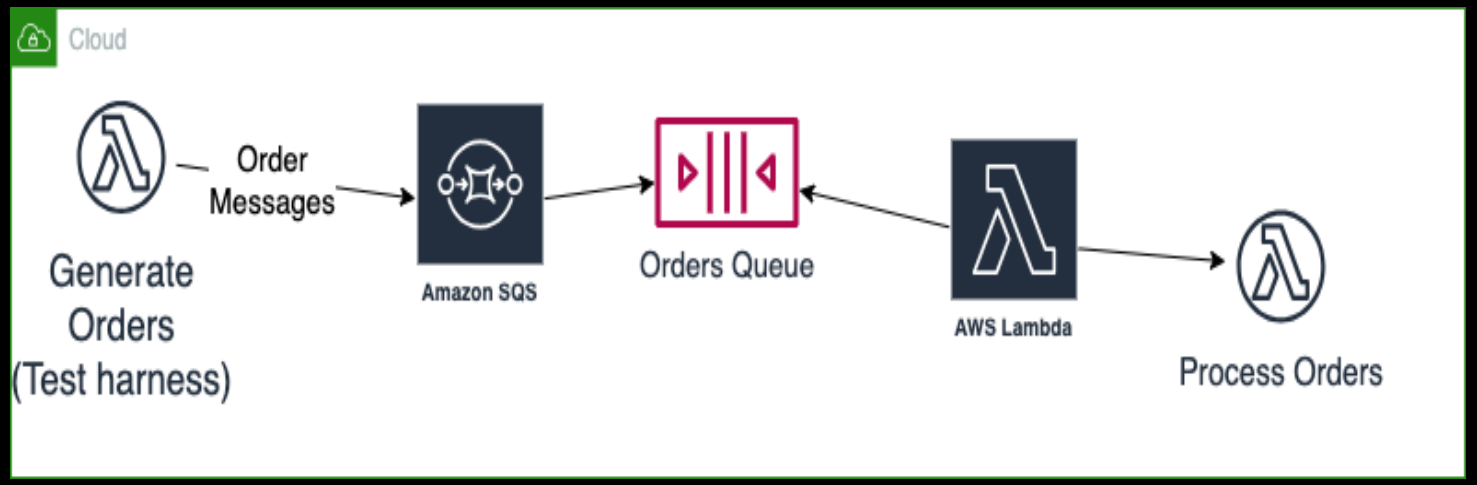
- When you consume messages from a queue, the SQS service samples a subset of its servers (based on a weighted random distribution) and returns messages from only those servers.

Lambda Consumer scaling



- Lambda service:
  - Polls SQS as it waits for messages to arrive.
  - Consumes messages in batches, starting at five concurrent batches with five functions at a time.
  - It adds up to 60 functions per minute, up to 1,000 functions, to consume large message volumes.

Demo



- CDK code – Generate Orders lambda function needs permission to send messages to a queue, i.e. ordersQueue.grantSendMessages(generateOrdersFn)

Demo - Generate Orders (Producer)

shared > TS types.d.ts > ...

```ts
export type Order = {
  customerName: string;
  customerAddress: string;
  items: string[];
};

export type BadOrder = Partial<Order>;
export type OrderMix = Order | BadOrder;
```

```ts
const orders: OrderMix[] = []
for (let i = 0 ; i < 10; i++) {
  orders.push({
    customerName: `User${i}`,
    customerAddress: "1 Main Street",
    items: [],

  })
}
```

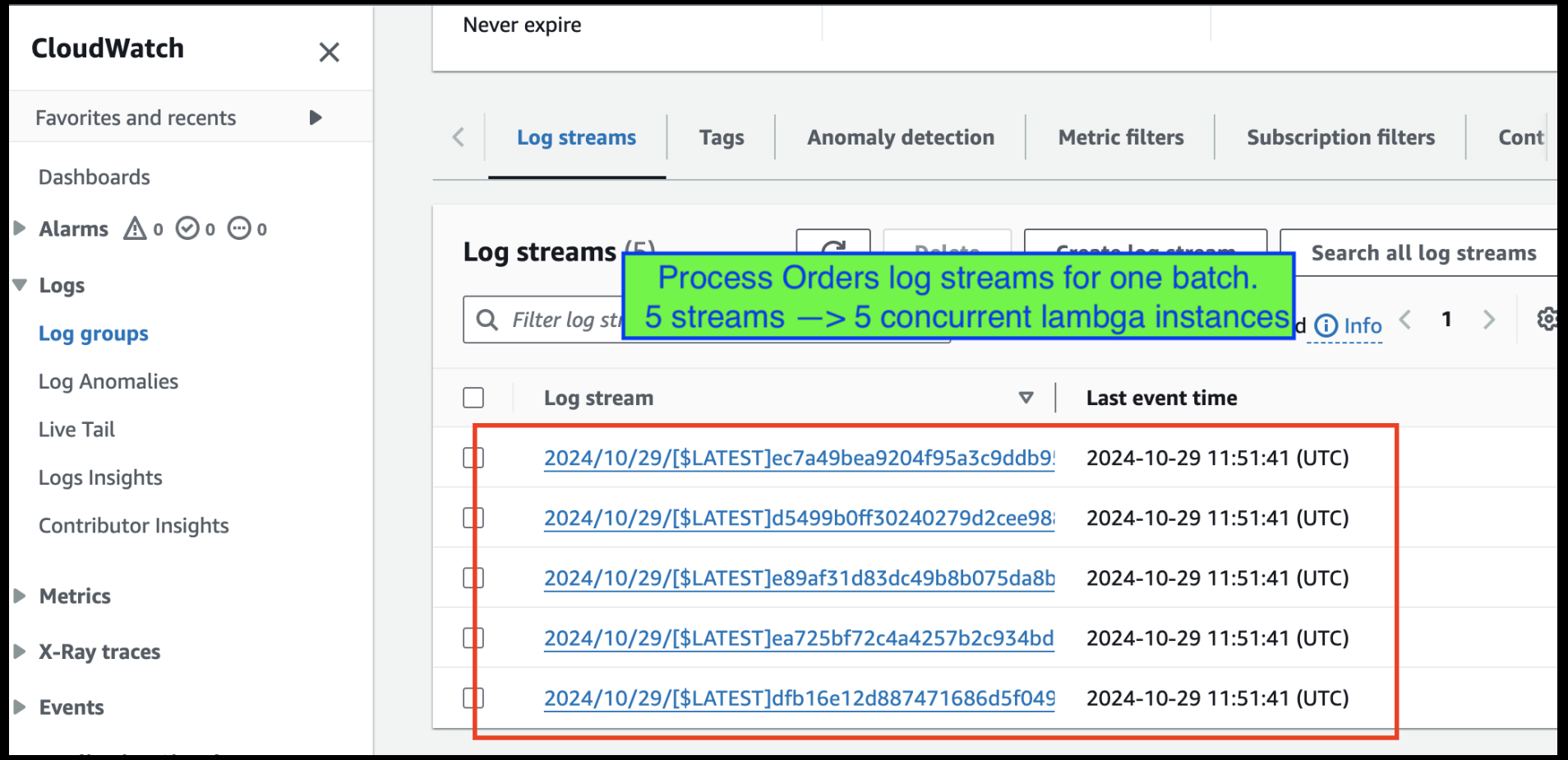
```ts
const client =
          new SQSClient({ region: "eu-west-1" });
      You, 3 minutes ago • Uncommitted changes
export const handler: Handler = async (event) => {
  try {
    const entries: SendMessageBatchRequestEntry[] =
     orders.map((order) => {
       return {
         Id: v4(),
         MessageBody: JSON.stringify(order),
       };
    });
    const batchCommandInput: SendMessageBatchCommandInput = {
      QueueUrl: process.env.QUEUE_URL, Entries: entries,
    };
    const batchResult = await client.send(
      new SendMessageBatchCommand(batchCommandInput)
    );
    return {
      statusCode: 200,
      headers: {
        "content-type": "application/json",
      },
      body: "All orders queued for processing",
    };
```

Demo – Process Orders (Consumer)

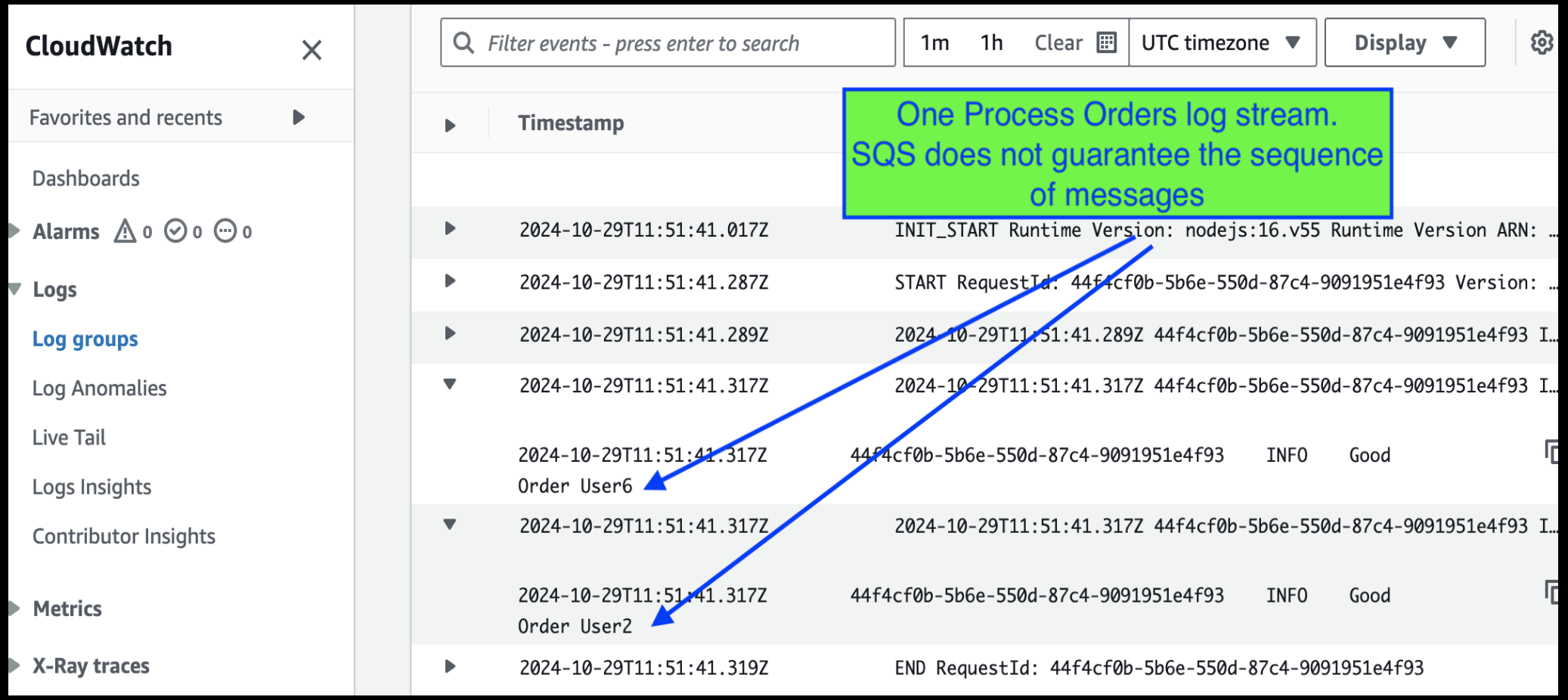
```
// Order Q processor

const ajv = new Ajv();
const isValidOrder = ajv.compile(schema.definitions["Order"] || {});
export const handler: SQSHandler = async (event) => {
  try {
    for (const record of event.Records) {
      const messageBody = JSON.parse(record.body);
      if (!isValidOrder(messageBody)  ) {
        throw new Error(" Bad Order");
      }
      // process good order
    }
  } catch (error) {
    throw new Error(JSON.stringify(error));
  }
};
```

Who handles the exception? (see later)

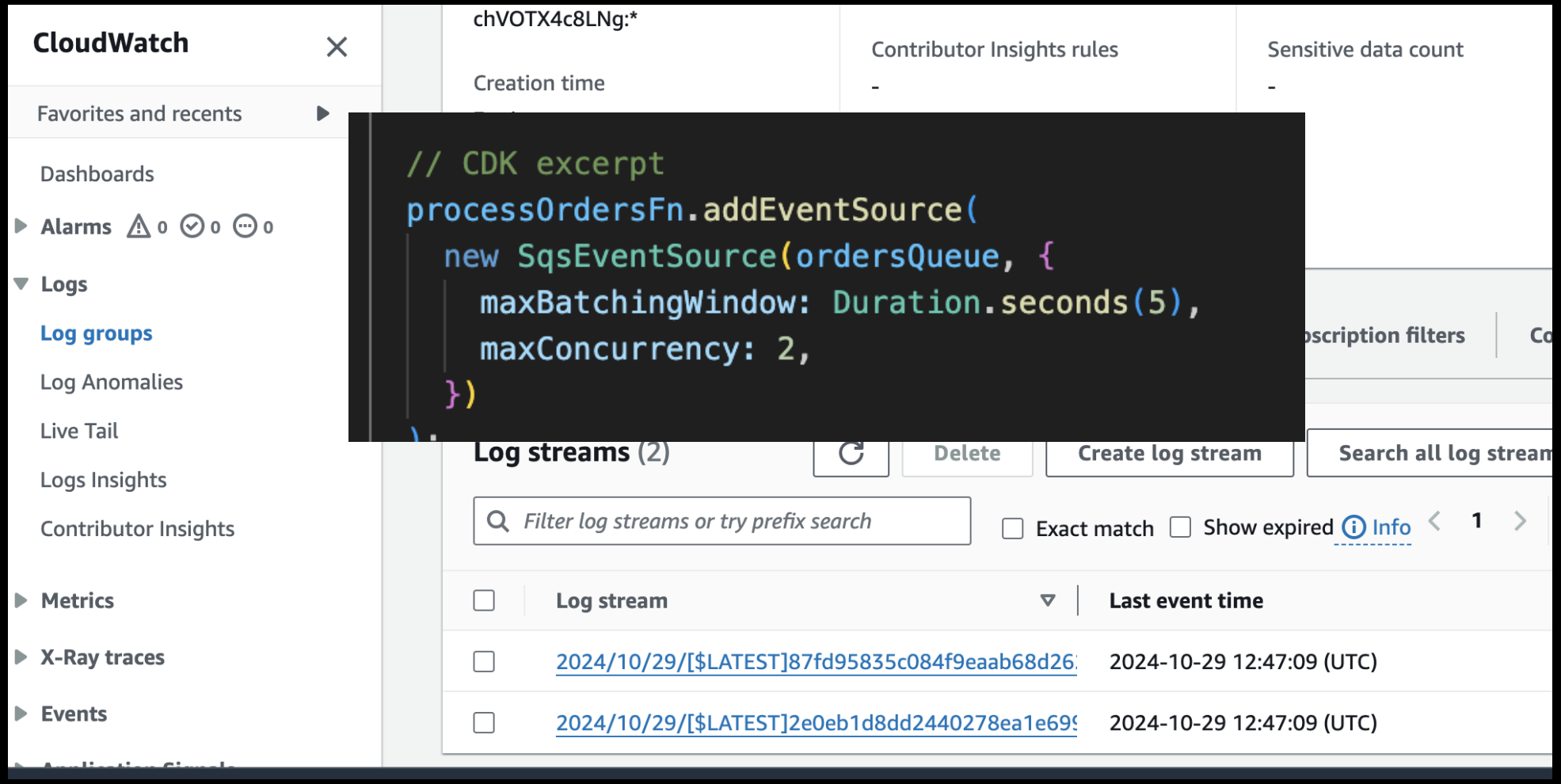
Demo – Lambda consumer scaling



Demo – Message order guarantee (not)

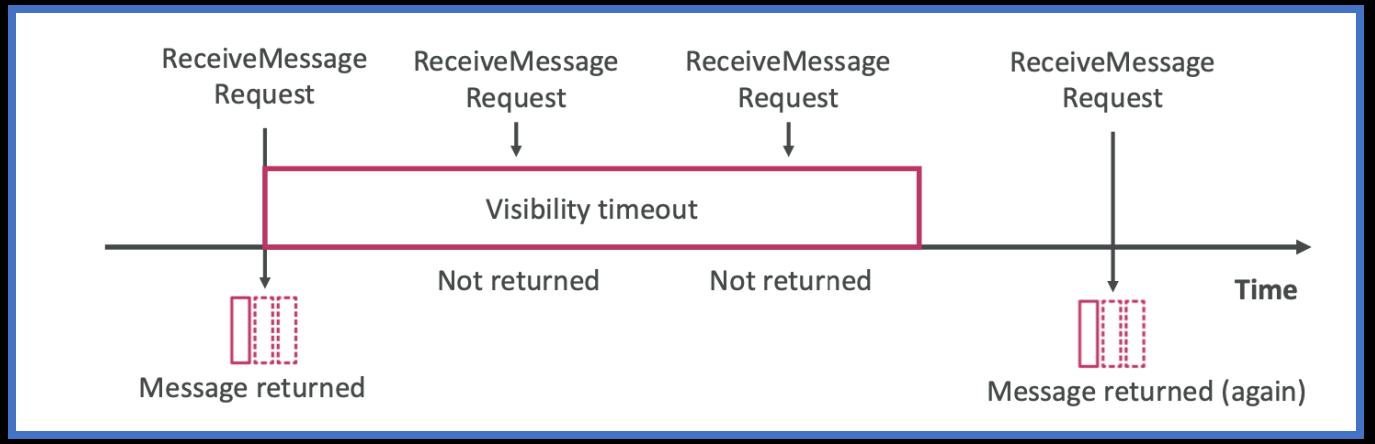


Demo – Controlling consumer concurrency.



Message Visibility.

- When a message is polled by a consumer, it remains in the queue but becomes invisible to other consumers.
  - The default “message visibility timeout” is 30 seconds.
  - => Consumer has 30 seconds to process (and delete) message. Otherwise, the message is “visible” again.



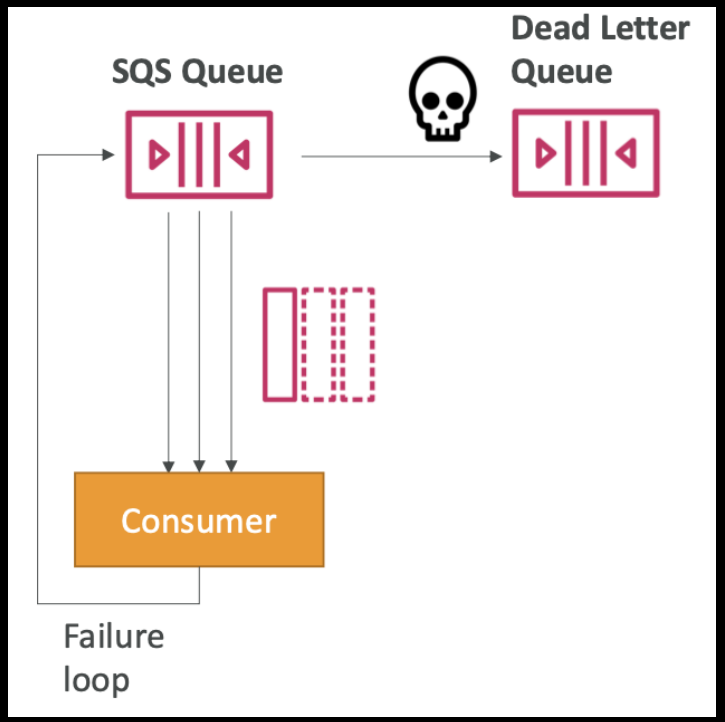
- If visibility timeout is too high (minutes/hours), and consumer crashes, re-processing is delayed.

If visibility timeout is too low, the message may be processed by more than one consumer.

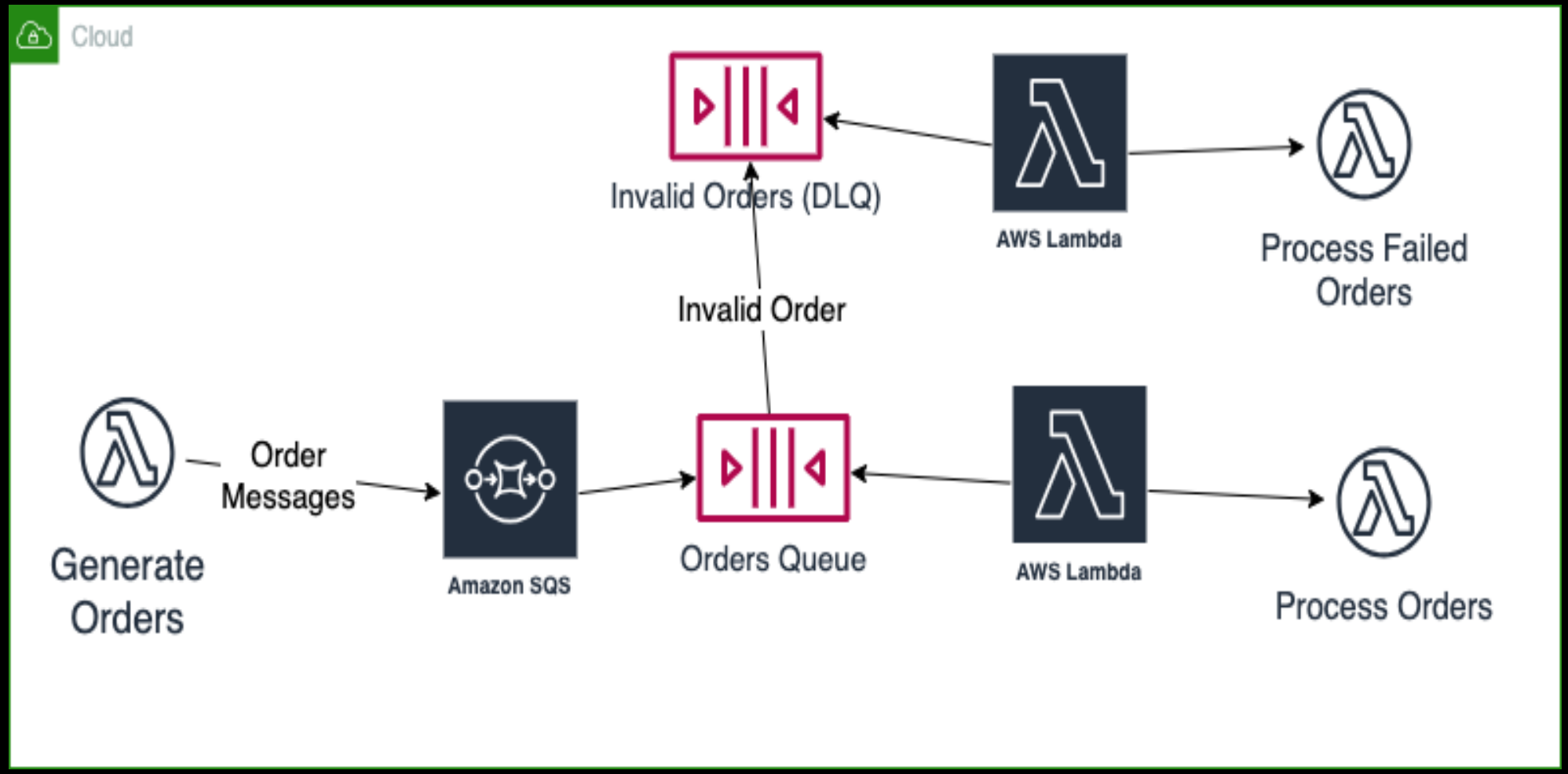
Dead Letter Queues (DLQs)

- If a consumer does not process a message batch within the visibility period (times out or throws exception), the batch is ‘returns to the queue’.
- Maximum Receives threshold - how many times a message is returned to the queue.
- After the threshold is exceeded, the message goes into a DLQ, if defined.

- Useful for debugging!
- Separate consumer required to process DLQ messages.



Demo – DLQ.


Cloud
Invalid Orders (DLQ)
AWS Lambda
Process Failed Orders
Invalid Order
Order Messages
Generate Orders
Amazon SQS
Orders Queue
AWS Lambda
Process Orders

Demo – Provision the DLQ .

```
// Relevant CDK code

const badOrdersQueue = new Queue(this, 'bad-orders-q');

const ordersQueue = new Queue(this, 'orders-queue', {
  deadLetterQueue: {
    queue: badOrdersQueue,
    // # of rejections by consumer (lambda function) before
    // message is transferred to DLQ
    maxReceiveCount: 1,
  },
});

// .... declare Lambda function resources .......

// Set SQS queues as Event sources for lambda functions
processOrdersFn.addEventSource(new SqsEventSource(ordersQueue))
failedOrdersFn.addEventSource(new SqsEventSource(badOrdersQueue));

// Grant function IAM rights to send messages.
ordersQueue.grantSendMessages(generateOrdersFn)
     You, 1 second ago • Uncommitted changes
```

Demo – Generate Orders (Producer).

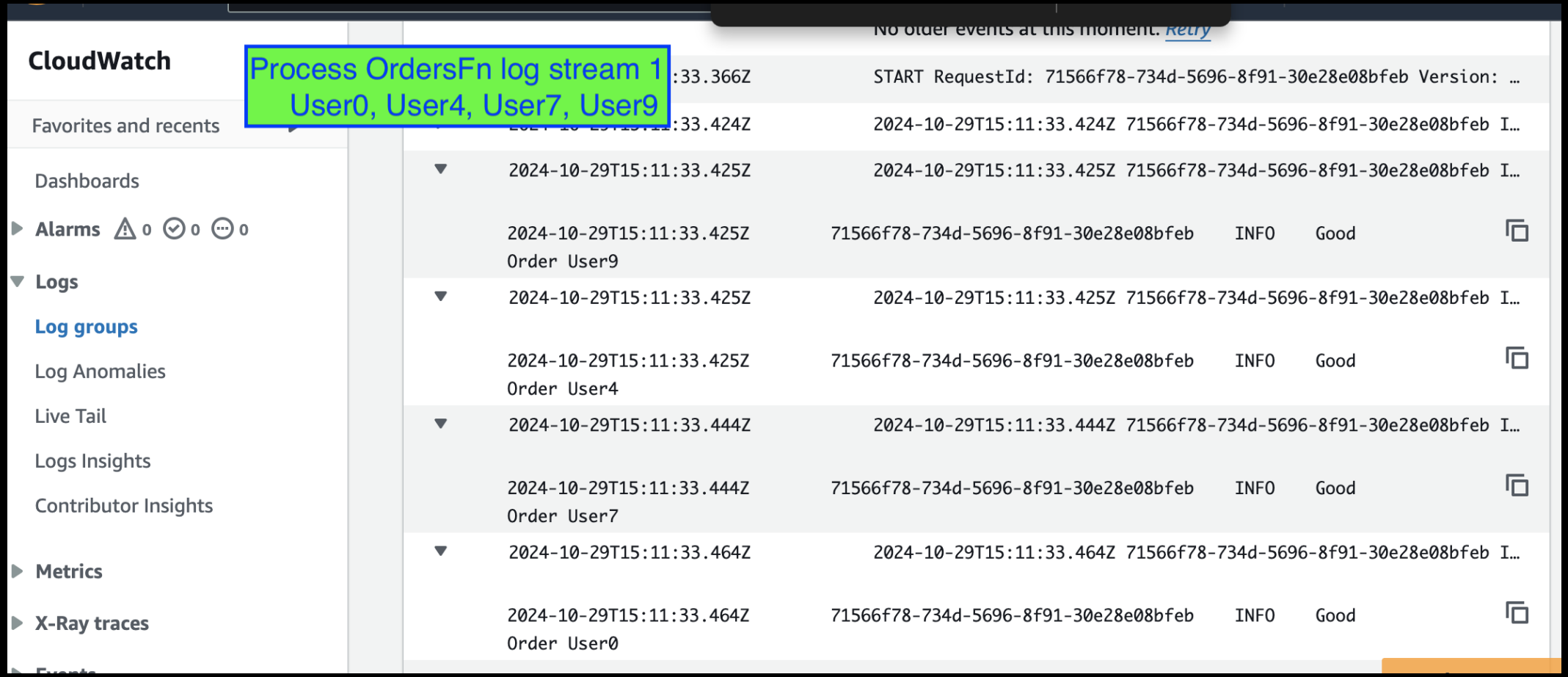
```
const orders: OrderMix[] = [];
for (let i = 0; i < 10; i++) {
  orders.push({
    customerName: `User${i}`,
    customerAddress: "1 Main Street",
    items: [],
  });
}

orders.splice(6, 0, {
  // No address property - Bad.
  customerName: "UserX",
  items: [],
});
```

Demo – Sample Execution

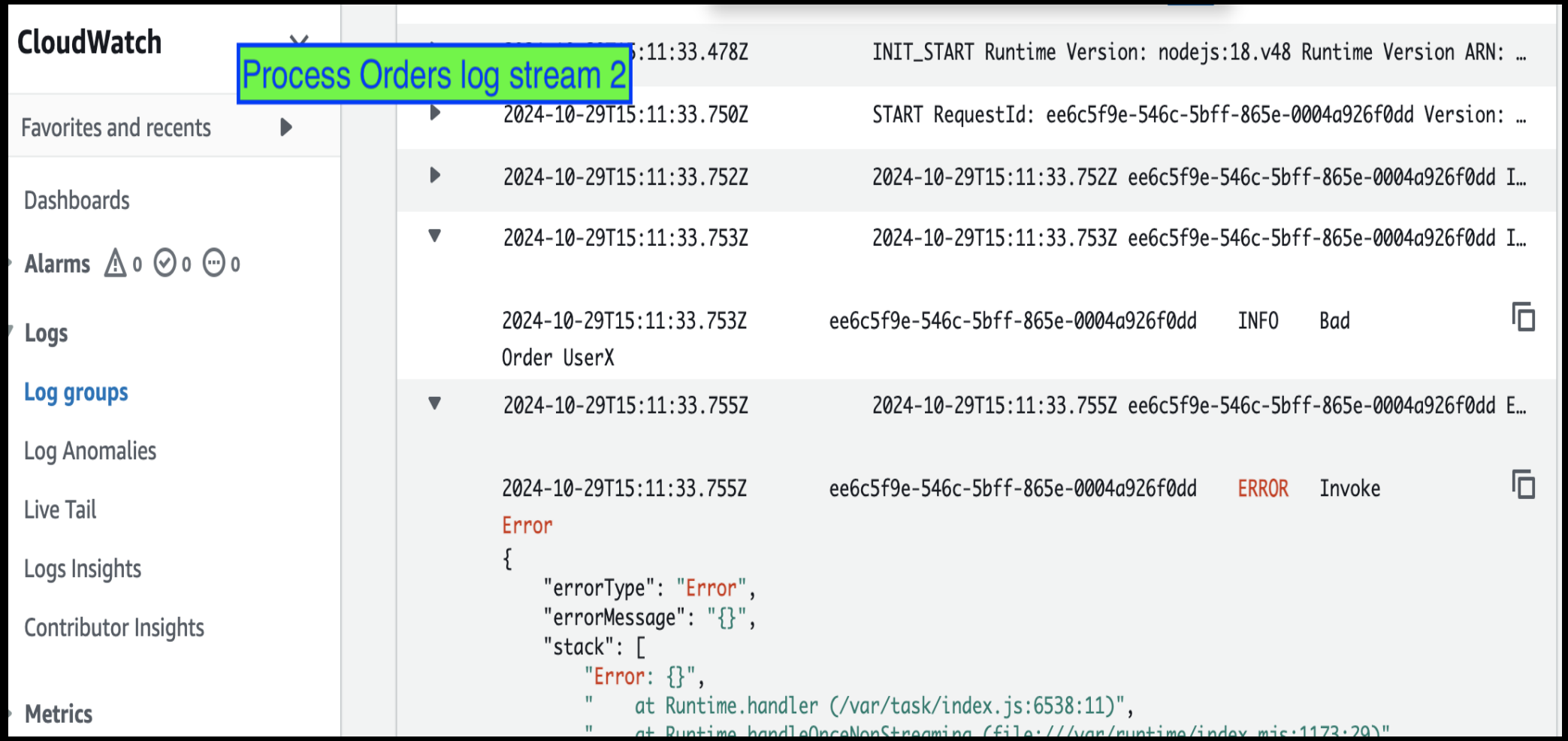
- Scenario:
  - Set the maxConcurrency of the Process Orders handler to 2.
  - Set the maximum receive count of the Orders q to 1 – batches containing a bad order are sent to DLQ after first failed processing attempt.

- Outcome:
  - Process Orders instance 1 → User0, User4, User7, User9.
  - Process Orders instance 2 → User1, User2, User3, User5, UserX, User8.
  - Bad Orders instance 1 → User1, User3, User8.
  - Bad Orders instance 2 → User2, User5, UserX.

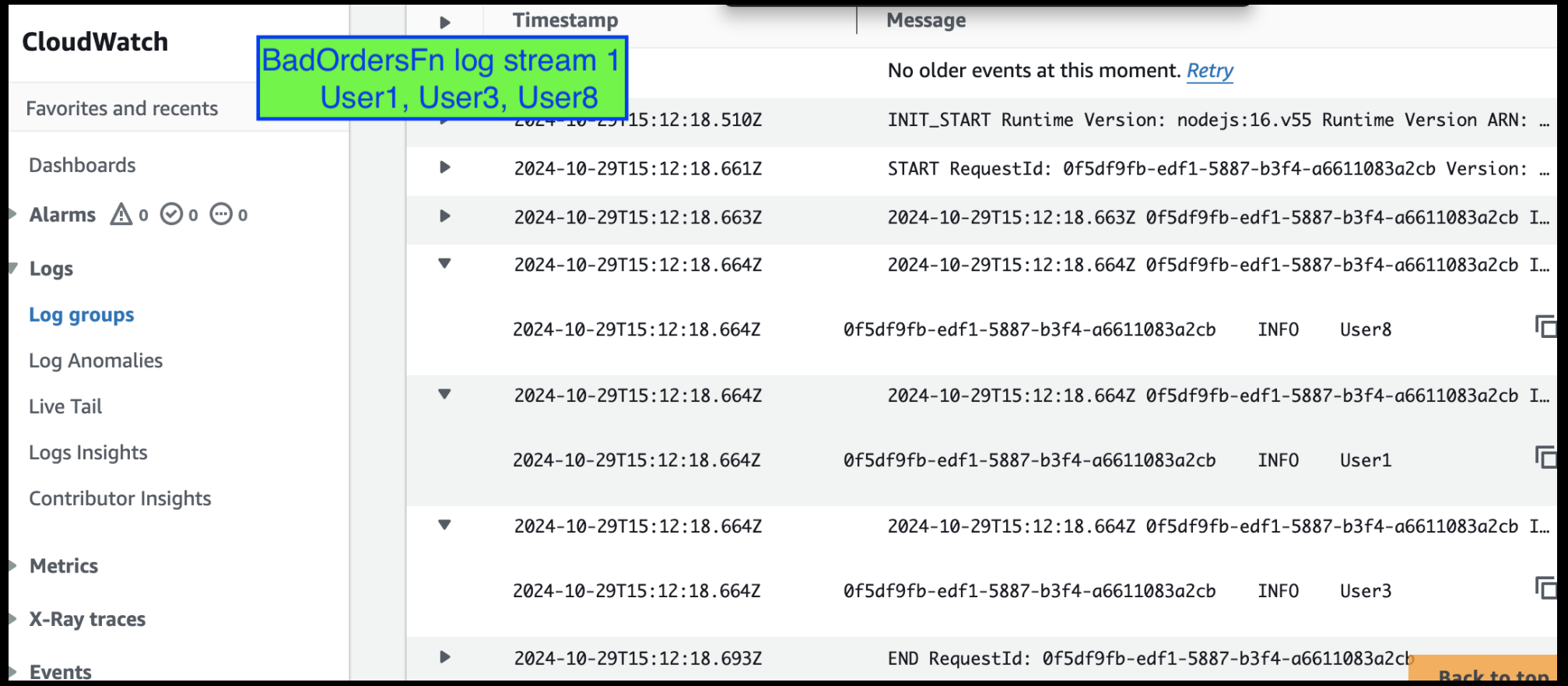
Demo – Sample Execution



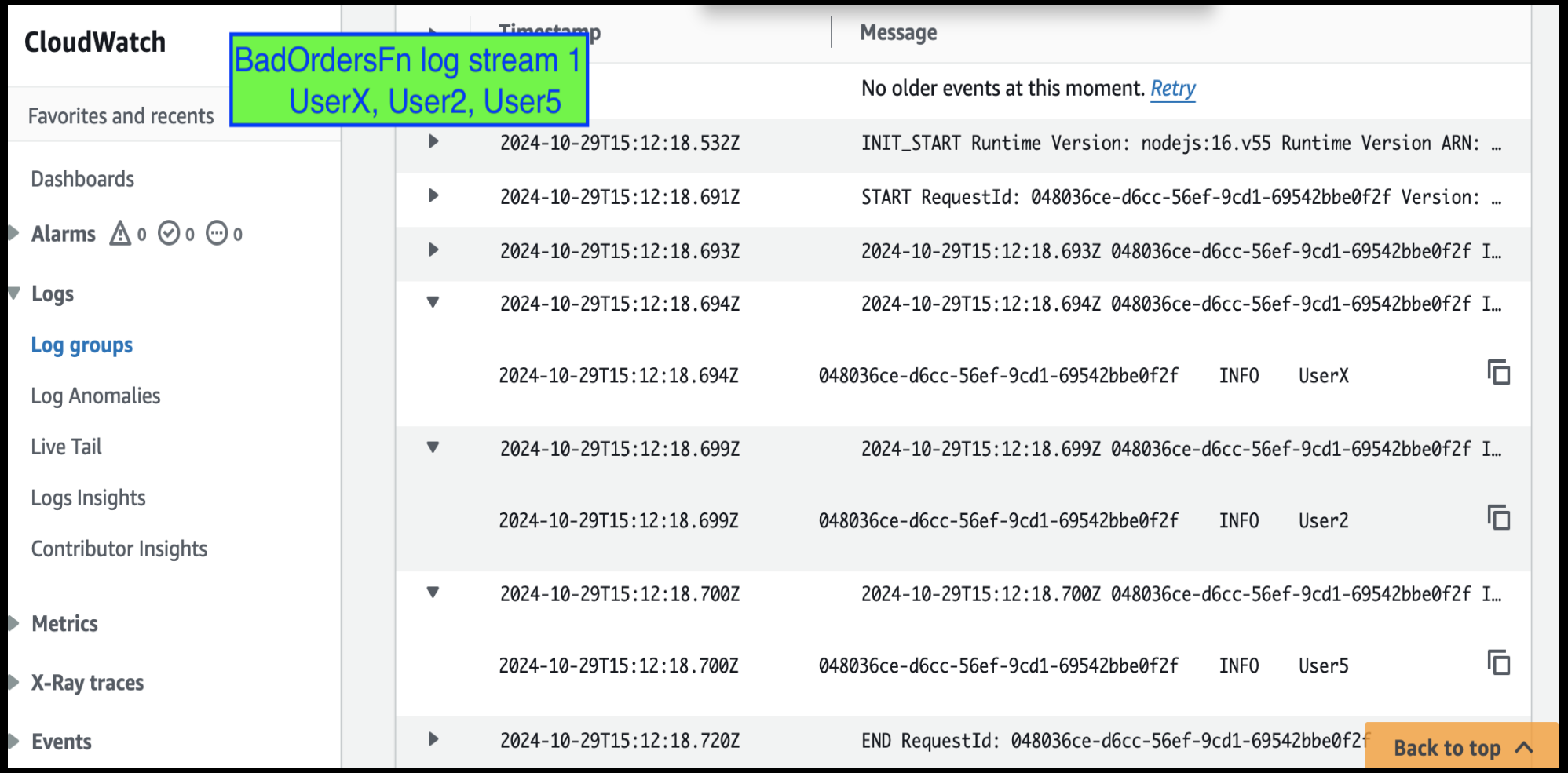
Demo – Sample Execution



Demo – Sample Execution



Demo – Sample Execution



SQS - Summary

- What:
  - Messaging service
  - Decoupling app compute components
- Why:
  - Decrease response time to client; Improve scalability.
- Actors: Producer and Consumer.
- Consumer polls the queue.
- Lambda function consumer.
  - Lambda service polls SQS; Scales handler instances
- Dealing with error cases:
  - Retries (Infinite by default)
  - Dead Letter Queue (DLQ)