How to succeed with serverless in a post-serverless world

NO MORE SERVERS!

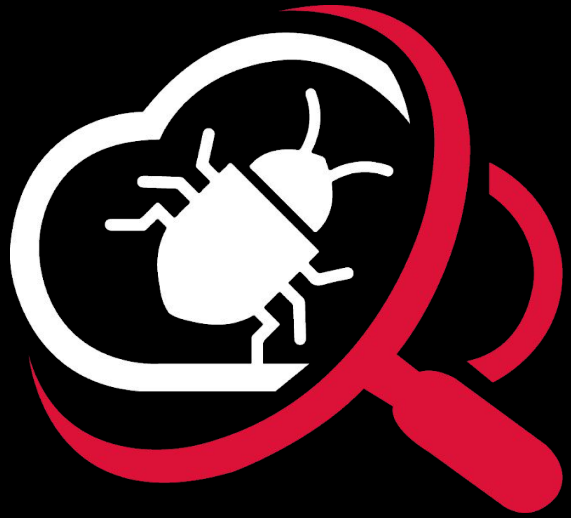
- I am Marko
- 5x AWS certificates, including 2x professional
- **I joined the war against servers 8 years.**
- Blog: www.serverlesslife.com
- I am the author of several open-source projects

SERVERLESSPY

LAMBDA LIVE DEBUGGER

CDK BOOSTER

NO MORE SERVERS!

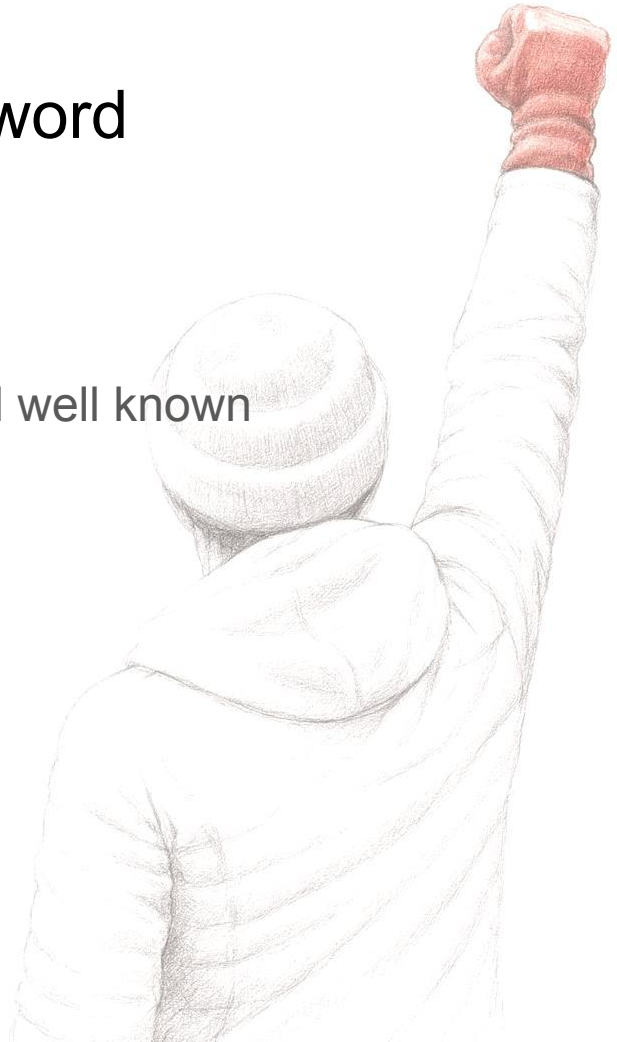


LAMBDA LIVE DEBUGGER

FREE OPEN-SOURCE TOOL FOR REMOTE DEBUGGING AWS LAMBDA FUNCTION

We are leaving in post-serverless word

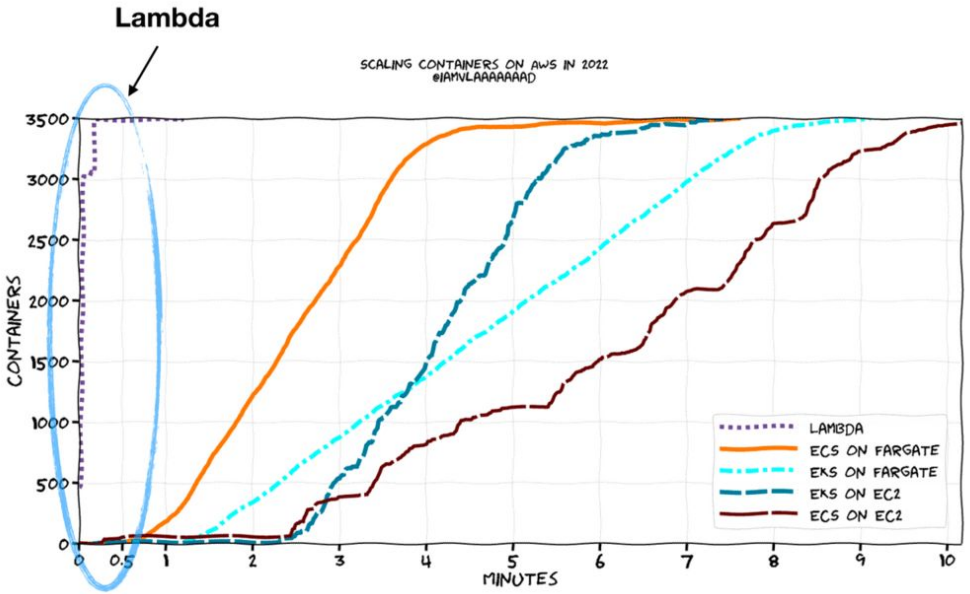
- Serverless is not hot topic anymore
- Serverless solutions are common
- All dark parts of serverless are resolved and well known



Why serverless?

- Automatic scaling (including scale-to-zero)

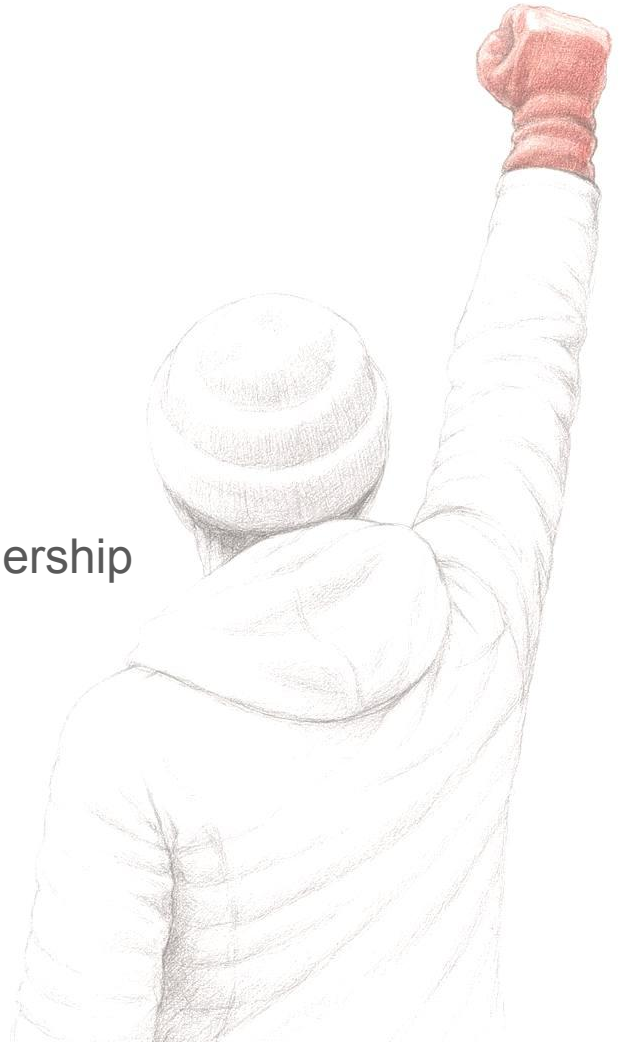
From 2023 - 1,000 new concurrent executions every 10 seconds



NO MORE SERVERS!

Why serverless?

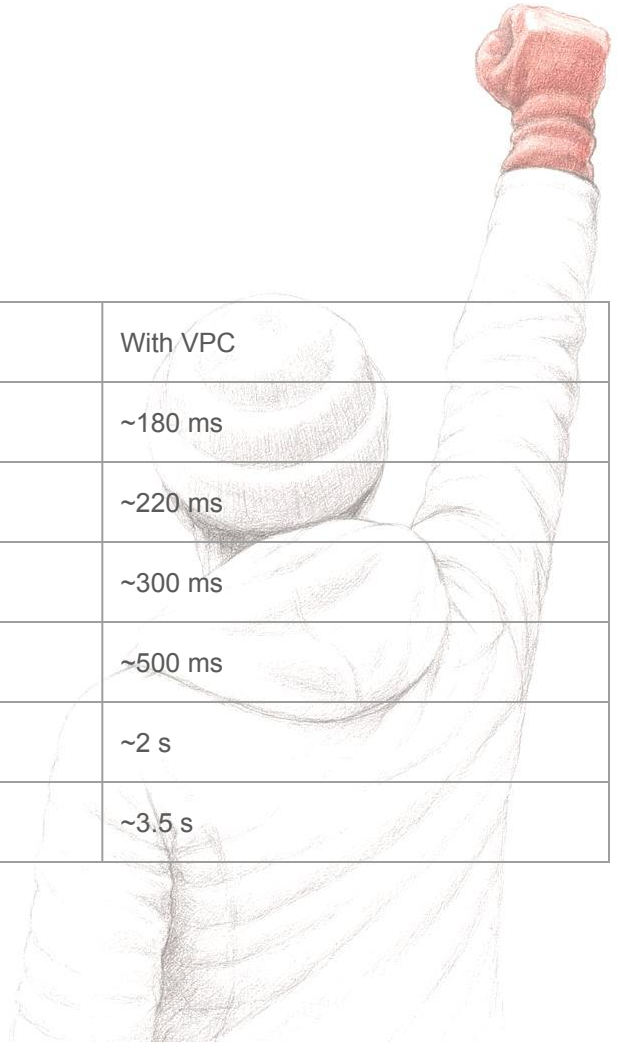
- Scale to zero
- Pay only for what you use
- Faster time-to-market
- Reduced operational burden
- Cheaper if you count in the total cost of ownership
- High availability and secure



Mythbusters

- Cold Starts

Runtime	No VPC	With VPC
Go	~50 ms ⚡	~180 ms
Rust	~70 ms ⚡	~220 ms
Node.js	~100 ms	~300 ms
Python	~200 ms	~500 ms
.NET	~1 s	~2 s
Java	~2 s 🐢	~3.5 s



NO MORE SERVERS!

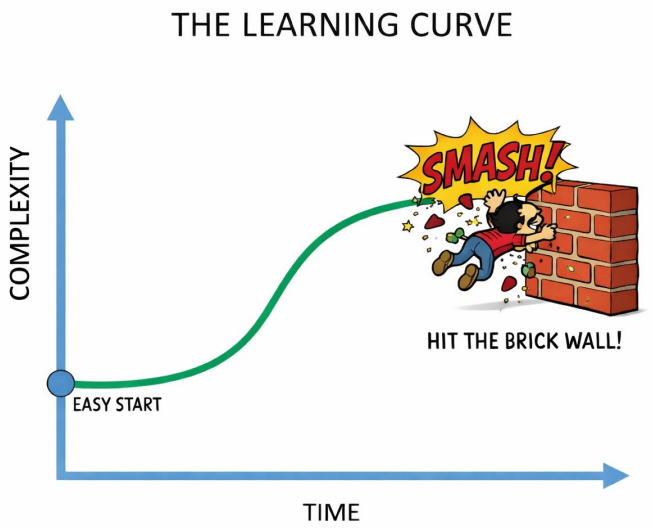
Mythbusters

- Lambda is too expensive
- Vendor lock-in
- Observability
- Lambda = Serverless
- SQL databases don’t work well with serverless

NO MORE SERVERS!

Why not serverless?

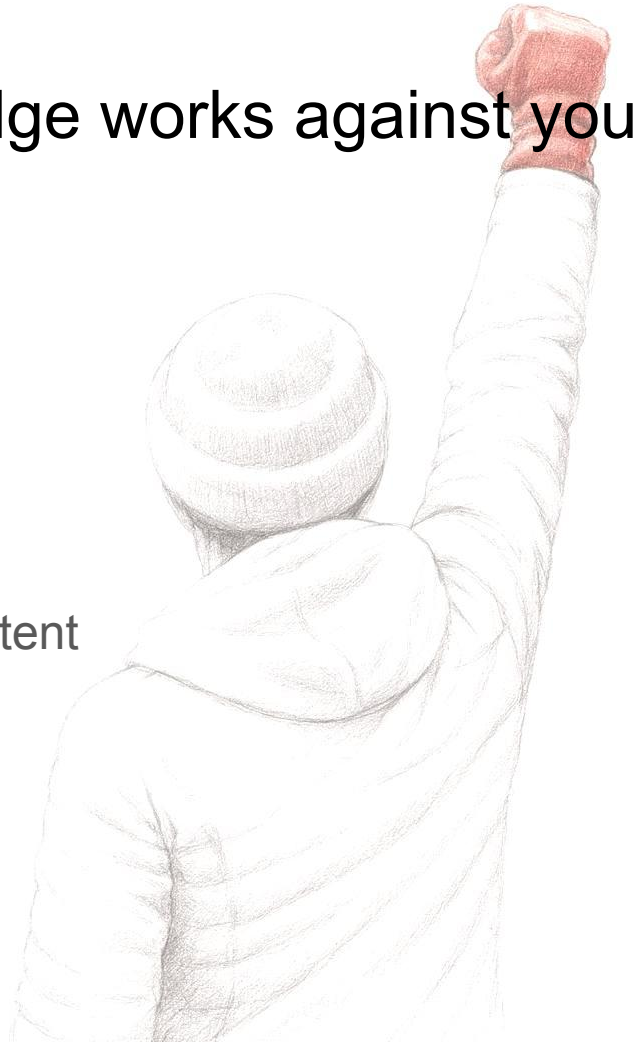
- Some technical limitations: 15 min timeout, …
- Steep learning curve



- Monolithic system, with a lot of libraries, ORM,... do not work well

Design patterns - your old knowledge works against you

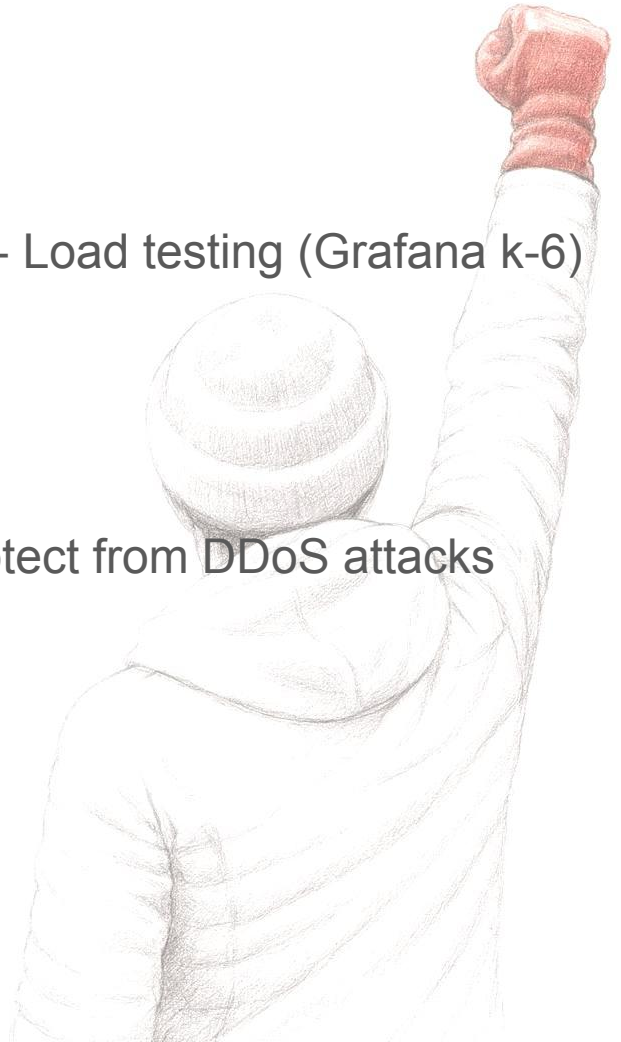
- Event-driven & asynchronous operations
- Eventual constancy
- Denormalization
- Transactions
- Retries & at-least-once delivery
- Design for failure → make functions idempotent



NO MORE SERVERS!

Do-s

1. Set AWS quotas (Lambda concurrency, …) - Load testing (Grafana k-6)
2. Set billing alarms
3. Watch out for recursive calls
4. Set Lambda memory
5. Set limits & WAF and other protection to protect from DDoS attacks
6. Have AWS account to develop on
7. Developers write IaC code
8. Developers should have access to billing



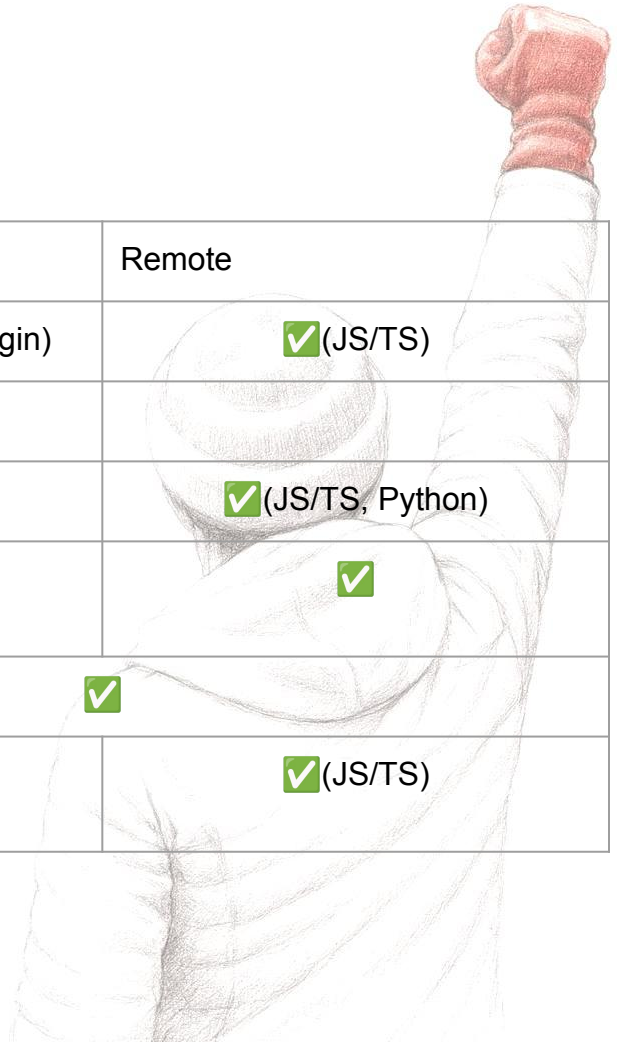
NO MORE SERVERS!

Development tools/frameworks

- Serverless framework
- AWS SAM
- AWS CDK
- Terraform
- SST
- …

NO MORE SERVERS!

Debugging



	Local	Remote
Serverless framework	✅ (serverless-offline plugin)	✅(JS/TS)
AWS SAM	✅	
SST		✅(JS/TS, Python)
AWS remote debugging with VsCode		✅
LocalStack	✅	
LAMBDA LIVE DEBUGGER		✅(JS/TS)

Databases

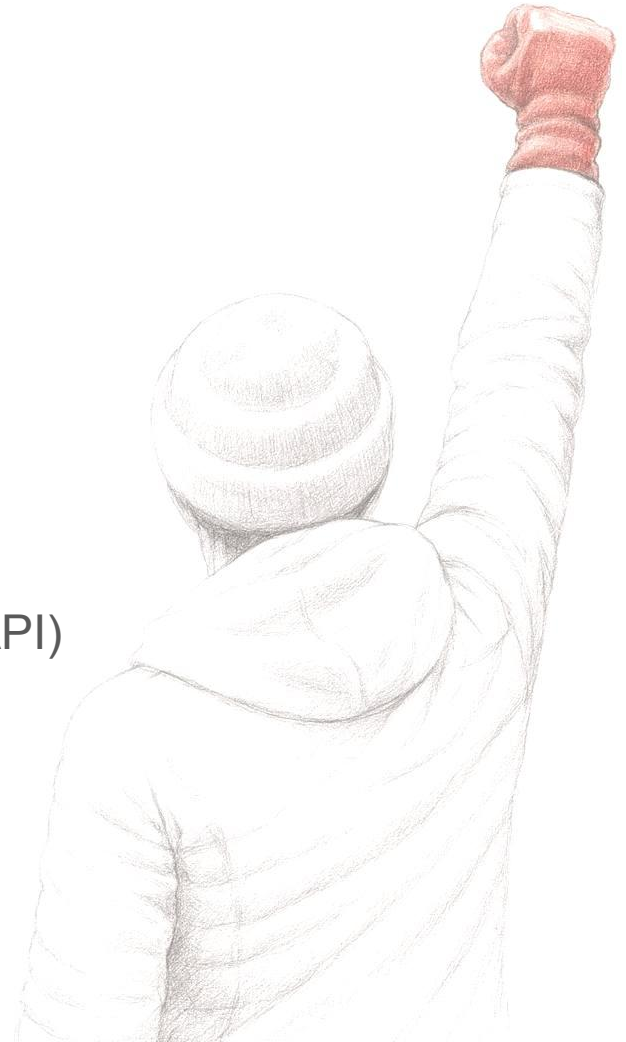
NOSQL:

- DynamoDB

  Single Table Design

SQL:

- RDS & Aurora (RDS Proxy)
- Aurora Serverless v2 (RDS Proxy or Data API)
- DSQL (no need for RDS Proxy)



NO MORE SERVERS!

New features

- Lambda Managed Instances
- Durable Lambda (long-running workloads)
- Amazon Bedrock AgentCore

NO MORE SERVERS!

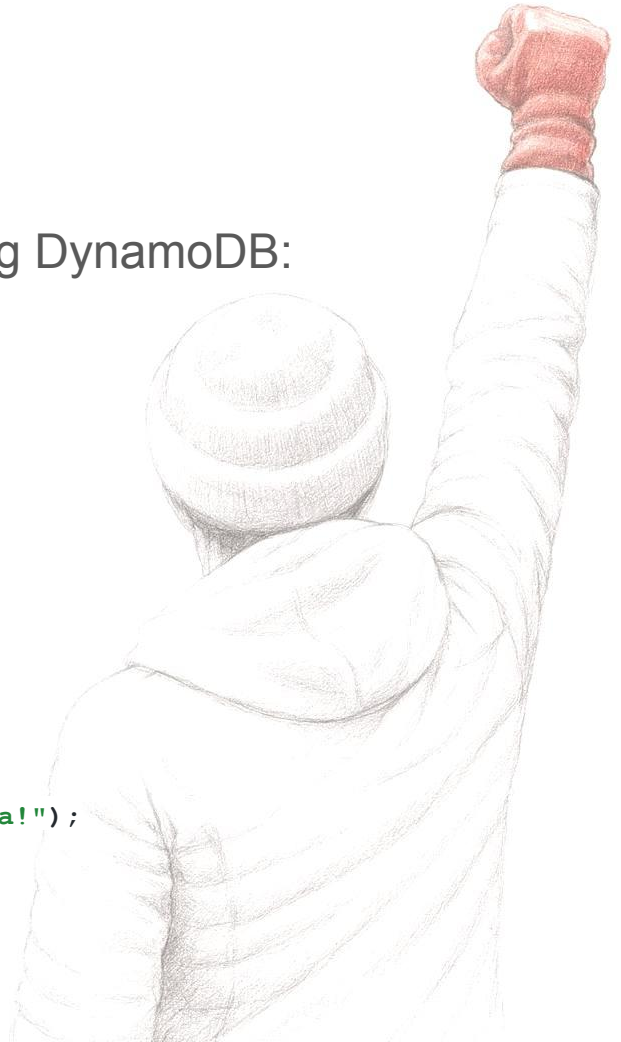
When actual behavior contradicts intuition

NO MORE SERVERS!

What is wrong with this code?

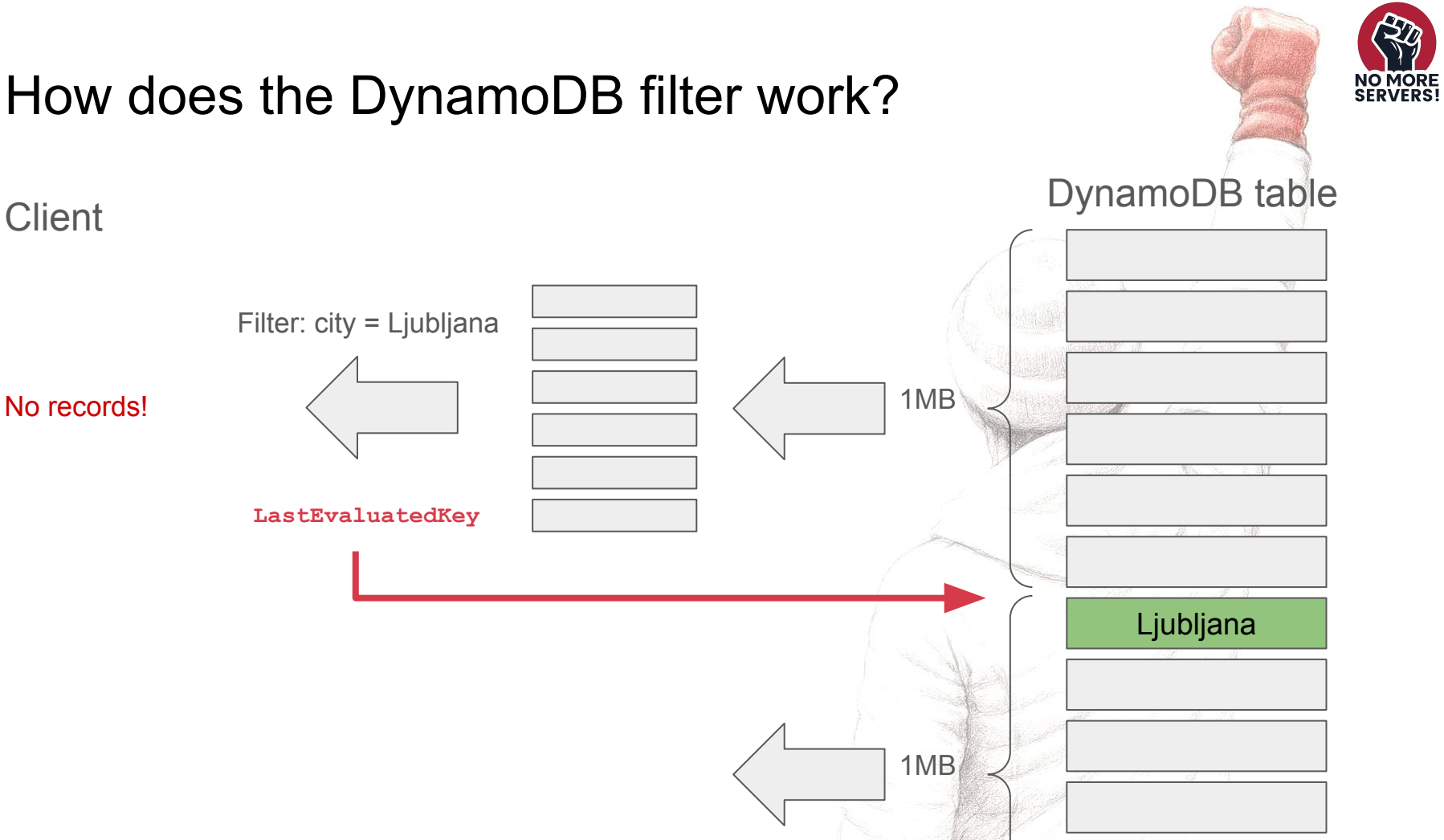
Searching for AWS User Group in Ljubljana using DynamoDB:

```
const result = await ddbDocClient.send(
  new ScanCommand({
    TableName: "usergroups",
    FilterExpression: "city = :city",
    ExpressionAttributeValues: { ":city": "Ljubljana" },
  })
);

if (result.Items && result.Items?.length > 0) {
  console.log("There is AWS User Group conference in Ljubljana!");
}
```



NO MORE SERVERS!

How does the DynamoDB filter work?



Fix #1

```
const allItems: any[] = [];
let lastEvaluatedKey: Record<string, NativeAttributeValue> | undefined;

do {
  const result = await ddbDocClient.send(
    new ScanCommand({
      TableName: "serverlessdays",
      FilterExpression: "city = :city",
      ExpressionAttributeValues: { ":city": "Milan" },
      ExclusiveStartKey: lastEvaluatedKey,
    })
  );

  if (result.Items) {
    allItems.push(...result.Items);
  }

  lastEvaluatedKey = result.LastEvaluatedKey;
} while (lastEvaluatedKey);

console.log(
  `Total found: ${allItems.length} ServerlessDays conferences in Milan`
);
```

NO MORE SERVERS!

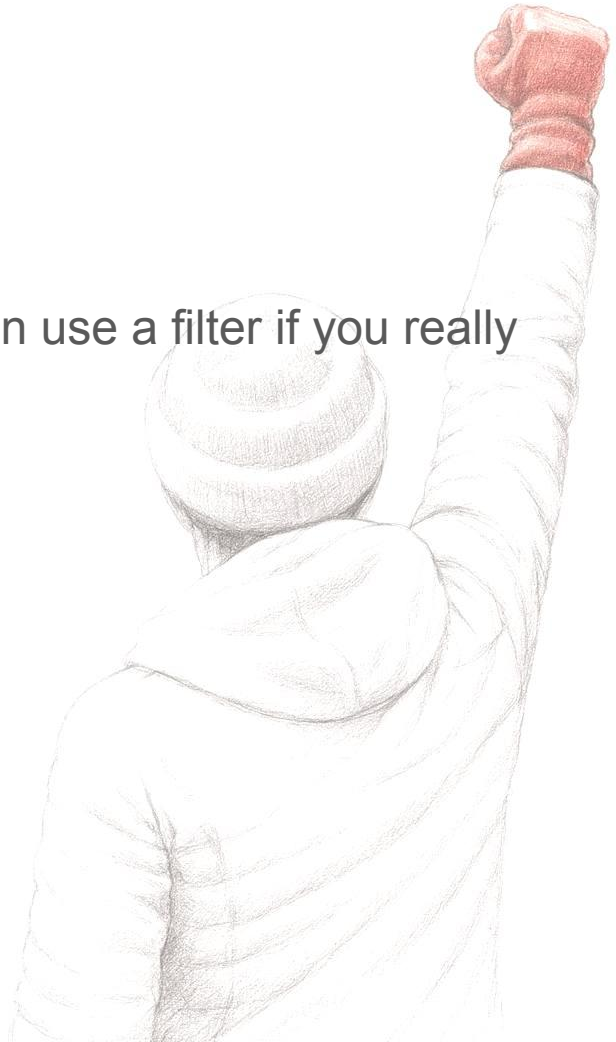
Fix #2

```
import { paginateScan } from "@aws-sdk/lib-dynamodb";

const paginator = paginateScan(
  { client: ddbDocClient },
  {
    TableName: "serverlessdays",
    FilterExpression: "city = :city",
    ExpressionAttributeValues: { ":city": "Milan" },
  }
);

for await (const page of paginator) {
  if (page.Items && page.Items?.length > 0) {
    console.log("There is ServerlessDays conference in Milan!");
    break;
  }
}
```

NO MORE SERVERS!

Fix #3

- Avoid using a filter on large datasets!
- Use indexes to narrow down results and then use a filter if you really need to.



NO MORE SERVERS!

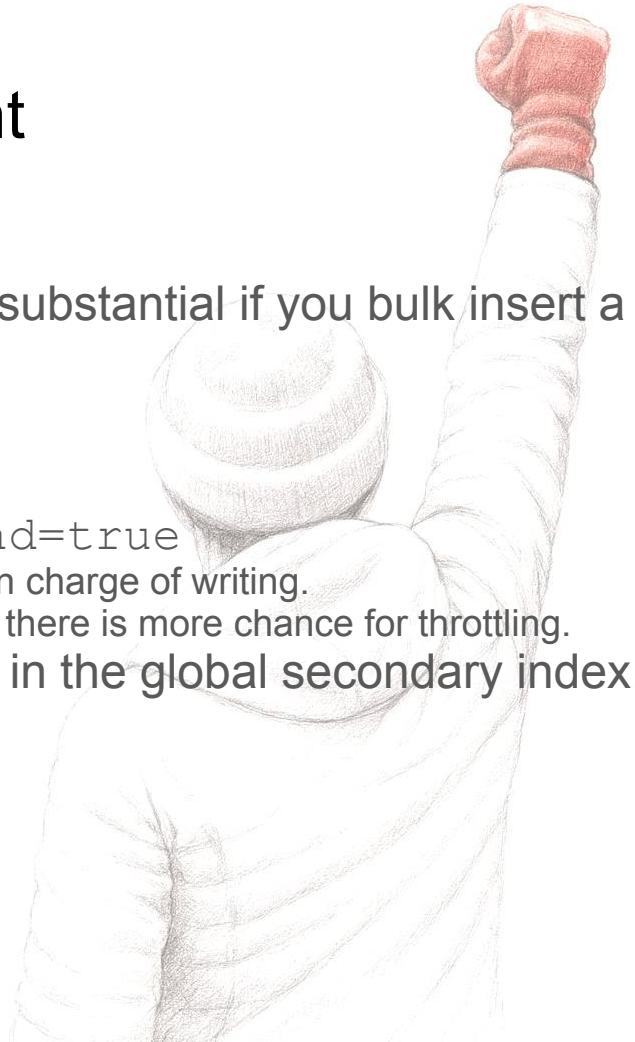
What is wrong with this code?

Writing and reading from DynamoDB:

```
await ddbDocClient.send(
  new PutCommand({
    TableName: "serverlessdays",
    Item: {
      PK: "conf-milan-2025",
      name: "ServerlessDays Milan 2025",
      city: "Milan",
    },
  })
);

const result = await ddbDocClient.send(
  new GetCommand({
    TableName: "serverlessdays",
    Key: {
      PK: "conf-milan-2025",
    },
  })
);
```

NO MORE SERVERS!

DynamoDB is eventually consistent



- Do not write and immediately read.
- Delay in the global secondary index can be substantial if you bulk insert a lot of data.

- Strongly consistent reads: `ConsistentRead=true`
  - You get a response from the leader replica that is in charge of writing.
  - Avoid if possible. It is slower, more expensive, and there is more chance for throttling.
- Strongly consistent reads are not supported in the global secondary index.

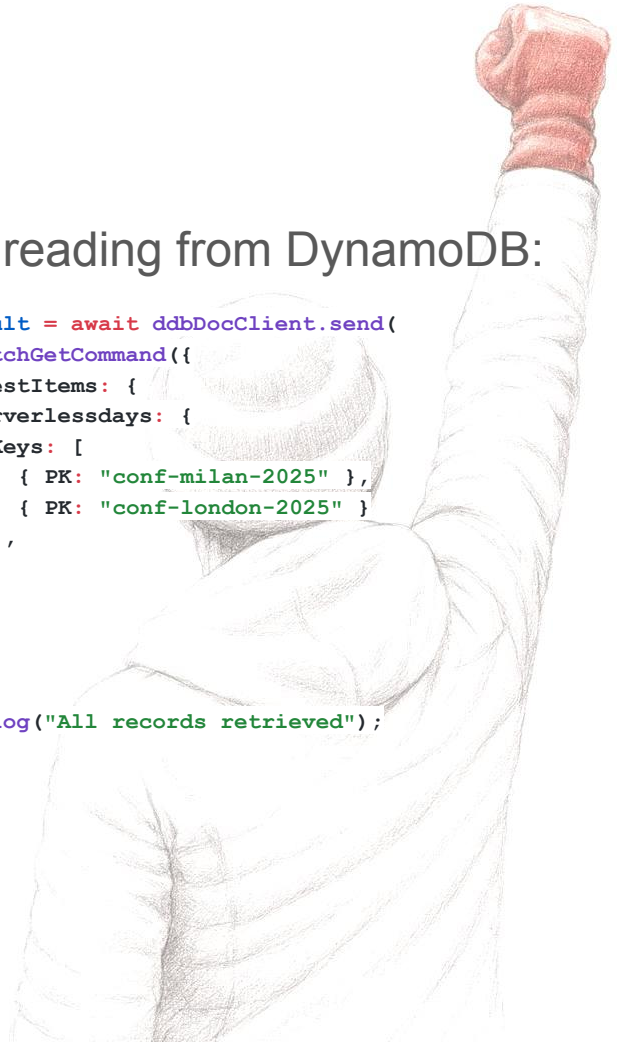
What is wrong with this code?

Batch writing to DynamoDB:

```
const writeResult = await ddbDocClient.send(
  new BatchWriteCommand({
    RequestItems: {
      serverlessdays: [{
          PutRequest: {
            Item: {
              PK: "conf-milan-2025",
              name: "ServerlessDays Milan",
              city: "Milan",
            }}
       },{
          PutRequest: {
            Item: {
              PK: "conf-london-2025",
              name: "ServerlessDays London",
              city: "London",
      }}}],
    },
  })
);

console.log("All records written");
```

Batch reading from DynamoDB:

```
const result = await ddbDocClient.send(
   new BatchGetCommand({
     RequestItems: {
       serverlessdays: {
         Keys: [
           { PK: "conf-milan-2025" },
           { PK: "conf-london-2025" }
         ],
       },
     },
   })
 );

 console.log("All records retrieved");
```



NO MORE SERVERS!

Fix

Batch writing to DynamoDB:

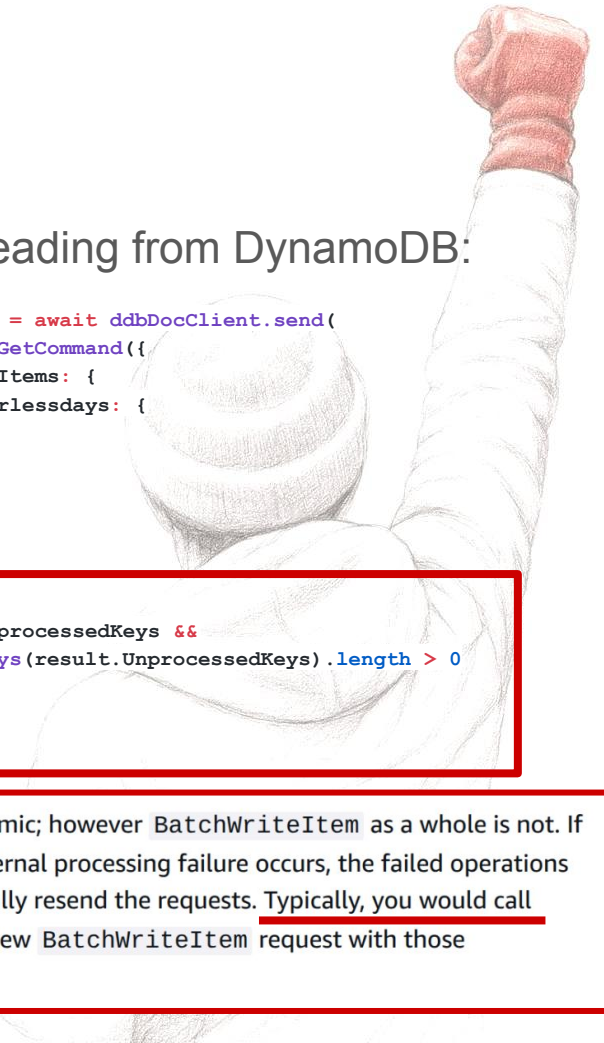
```
const writeResult = await ddbDocClient.send(
  new BatchWriteCommand({
      ...
    },
  })
);
```

```
if (
  writeResult.UnprocessedItems &&
  Object.keys(writeResult.UnprocessedItems).length > 0
) {
  // retry
}
```

Batch reading from DynamoDB:

```
const result = await ddbDocClient.send(
  new BatchGetCommand({
    RequestItems: {
      serverlessdays: {
        ...
      },
    },
  })
);
```

```
if (
  result.UnprocessedKeys &&
  Object.keys(result.UnprocessedKeys).length > 0
) {
  // retry
}
```



The individual `PutItem` and `DeleteItem` operations specified in `BatchWriteItem` are atomic; however `BatchWriteItem` as a whole is not. If any requested operations fail because the table's provisioned throughput is exceeded or an internal processing failure occurs, the failed operations are returned in the `UnprocessedItems` response parameter. You can investigate and optionally resend the requests. Typically, you would call `BatchWriteItem` in a loop. Each iteration would check for unprocessed items and submit a new `BatchWriteItem` request with those unprocessed items until all items have been processed.

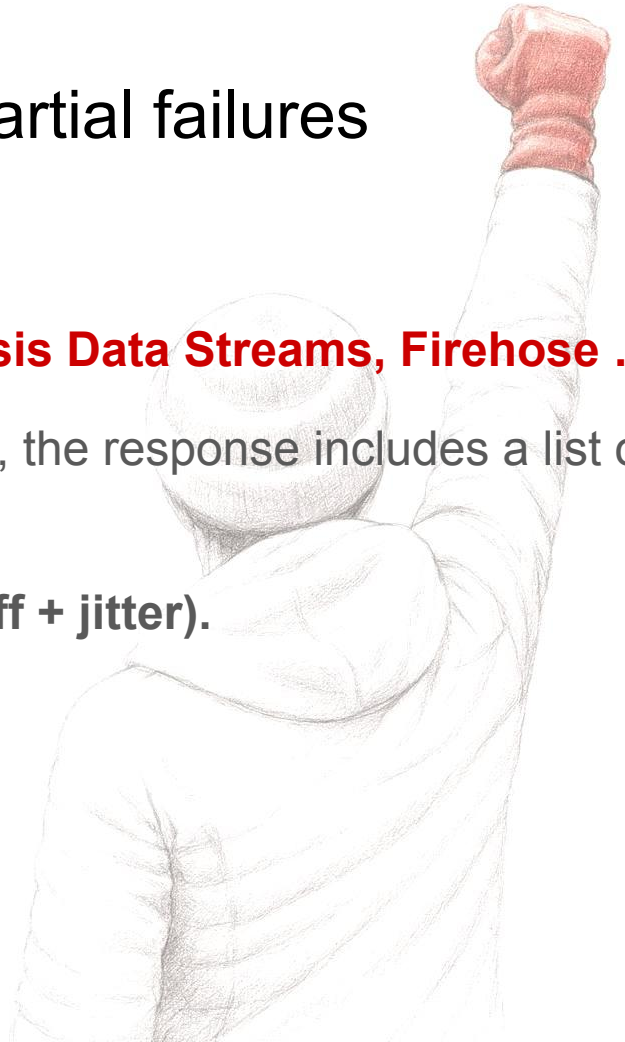
AWS SDK batch operations with partial failures

Batch operations in

DynamoDB, SQS, SNS, EventBridge, Kinesis Data Streams, Firehose …

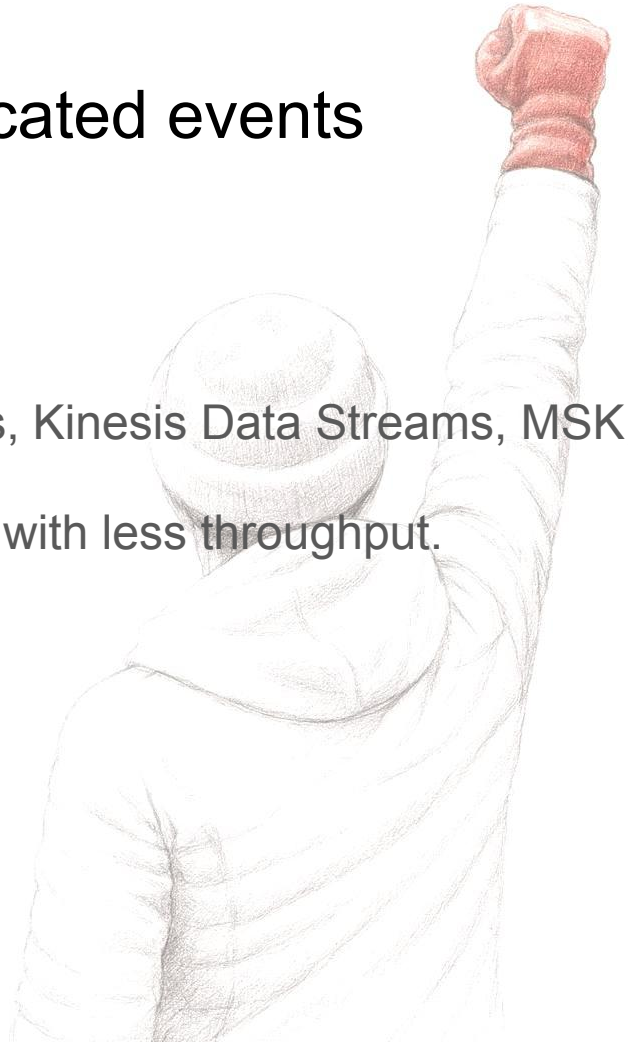
Partial failures don’t throw exceptions — instead, the response includes a list of failed items.

You must retry them (with exponential backoff + jitter).



NO MORE SERVERS!

Event-driven systems: order, duplicated events



Order

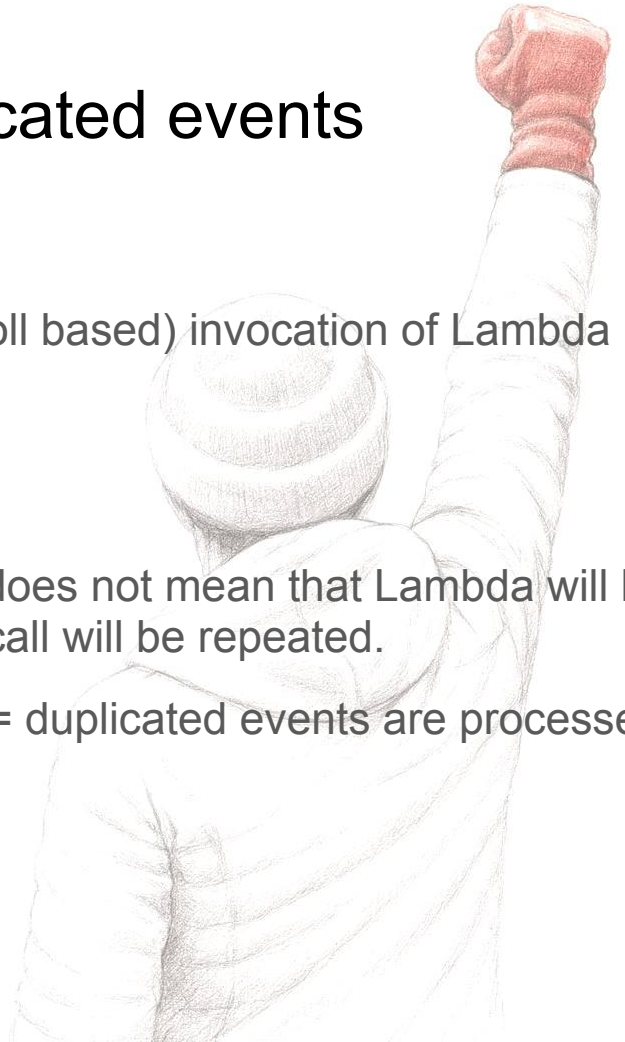
Services that guarantee order:

- SQS **FIFO**, SNS **FIFO**, DynamoDB Streams, Kinesis Data Streams, MSK, EventBridge Pipes when reading from FIFO
- SQS FIFO, SNS FIFO are more expensive, with less throughput.

Services that do not guarantee order:

- SQS, SNS, EventBridge… all the rest

Event-driven systems: order, duplicated events



NO MORE SERVERS!

Duplicated events

Services that guarantee ASYNCHRONOUSLY (or poll based) invocation of Lambda exactly once:

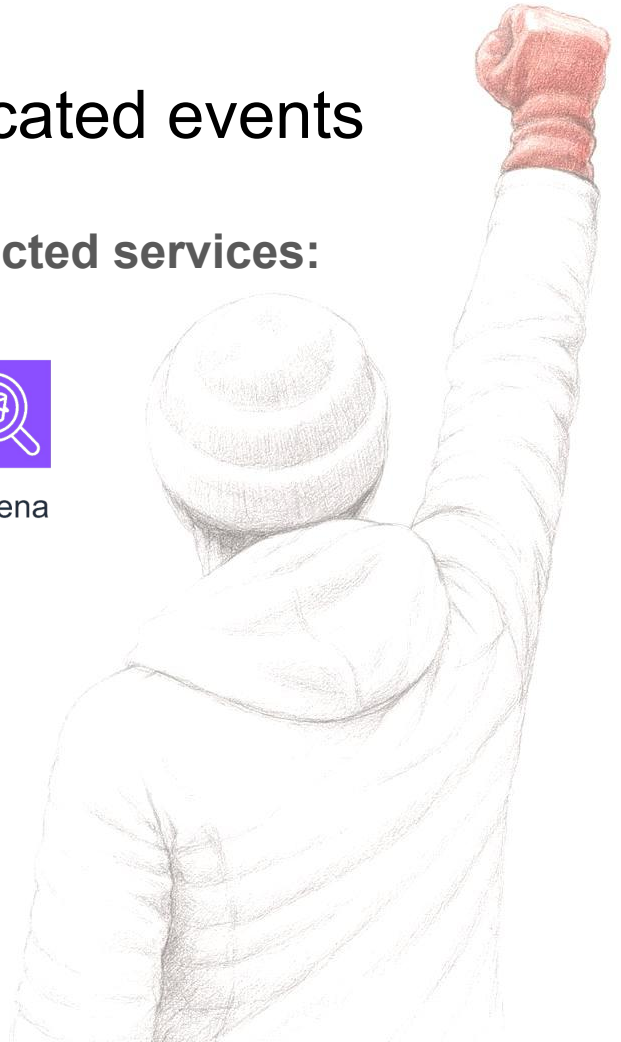
NONE!

SQS FIFO, SNS FIFO offer de-duplication, but that does not mean that Lambda will be triggered only once. If Lambda fails or time out, the call will be repeated.

You should design the system to be **IDEMPOTENT** = duplicated events are processed only once.

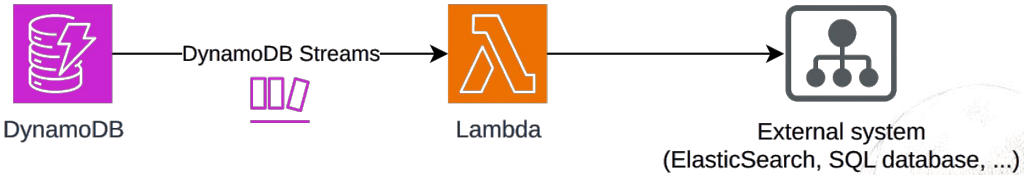
Event-driven systems: order, duplicated events

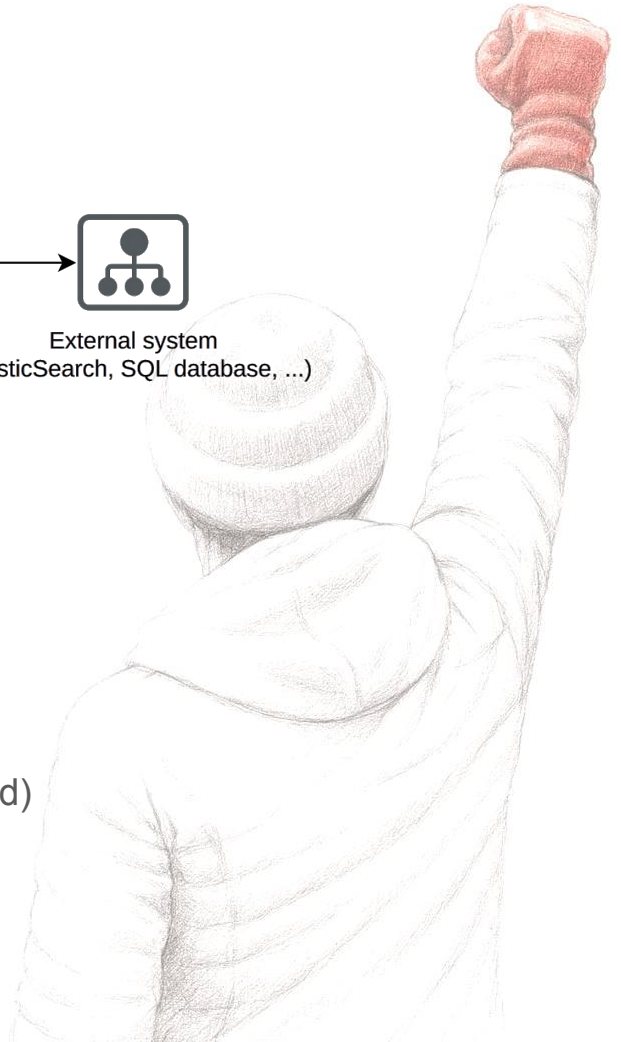
Duplicate events can occur even in less expected services:



NO MORE SERVERS!

Transactional Outbox pattern





Why it is great?

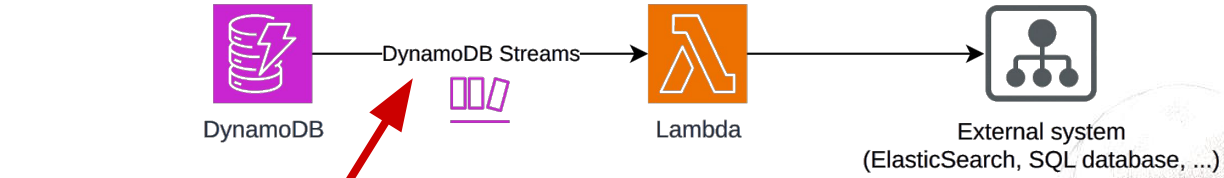
- Atomic & reliable
- Scalable

What would happen if:

- external system fails, is overloaded, too slow?
- poison pill message (a message that cannot be processed)

→ **DynamoDB Streams will retry, but …**

Transactional Outbox pattern



Records are lost after 24 hours

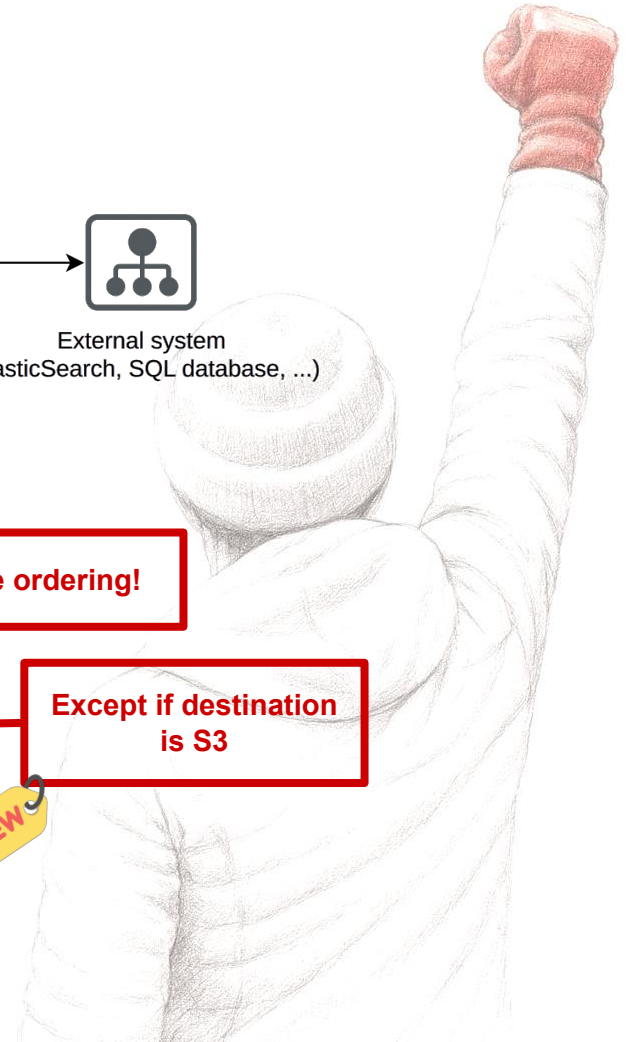
You lose ordering!

Does not contain record, just pointer to data in the stream.

Stream data disappears after 24 hours.

Except if destination is S3

NEW



NO MORE SERVERS!

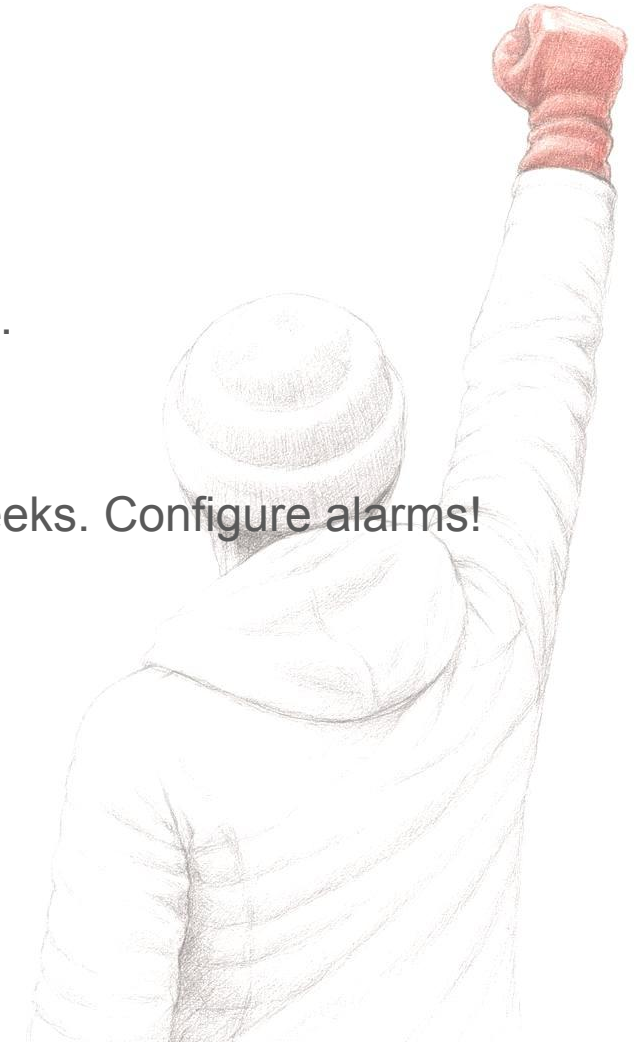
Dead-Letter Queue (DLQ) pattern

- DLQ = storage for failed message
- Usually SQS. Does not have to be a queue.

Issues:

- Messages in SQS are stored for up to 2 weeks. Configure alarms!
- Do not miss alarm message!

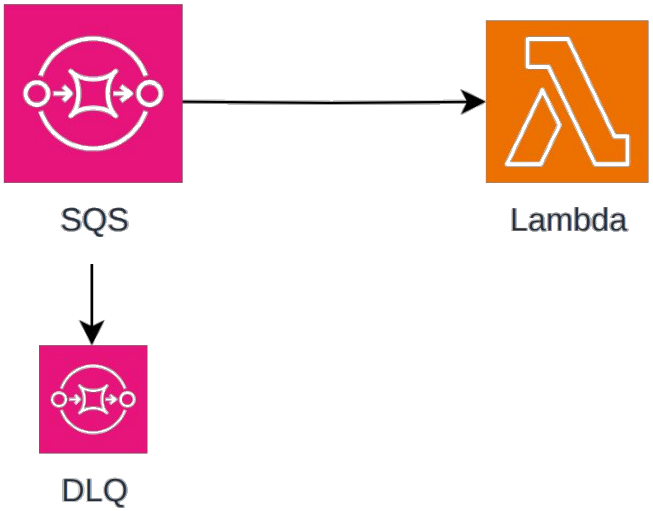
Challenge how to configure DLQ!



NO MORE SERVERS!

Dead-Letter Queue (DLQ)

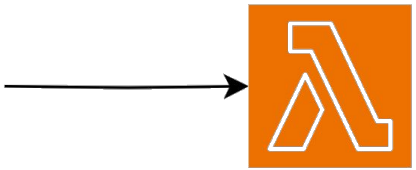
SQS



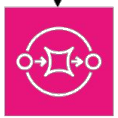
NO MORE SERVERS!

Dead-Letter Queue (DLQ)

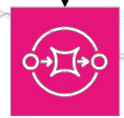
Async Lambda



Lambda



DLQ
(SQS, SNS)

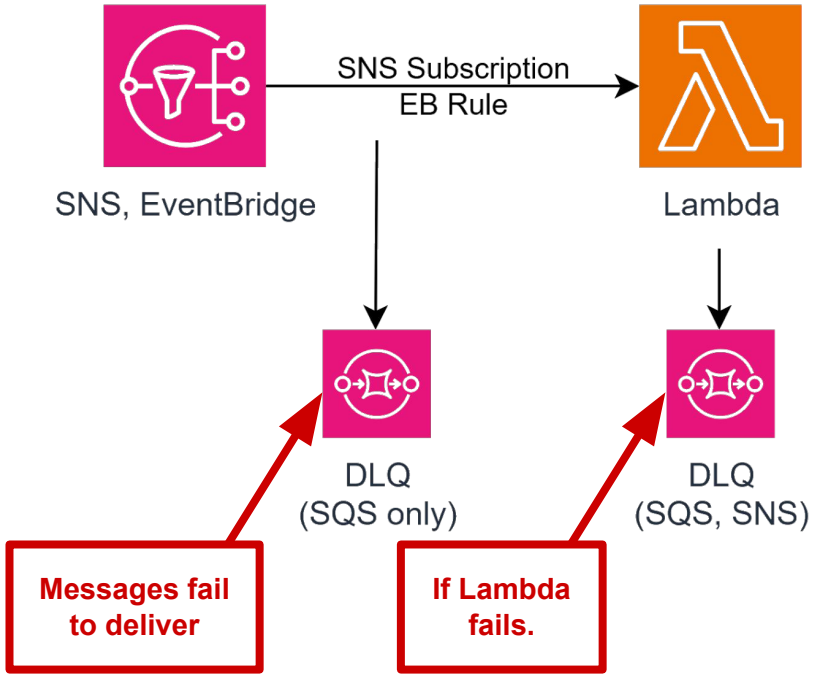


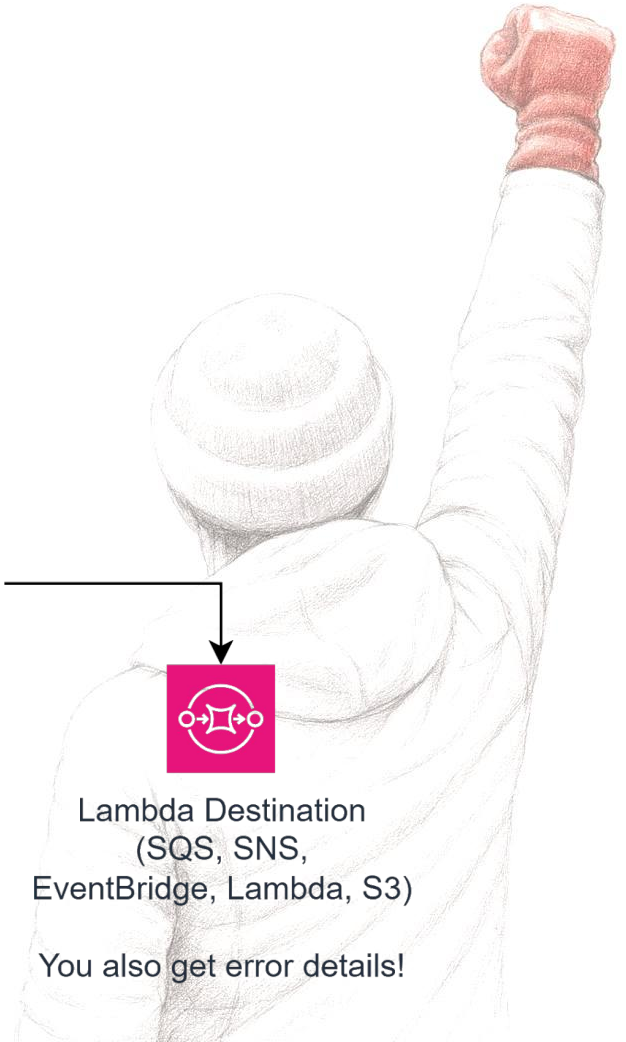
Lambda Destination
(SQS, SNS,
EventBridge, Lambda, S3)

You also get error details!

Dead-Letter Queue (DLQ)

SNS, EventBridge

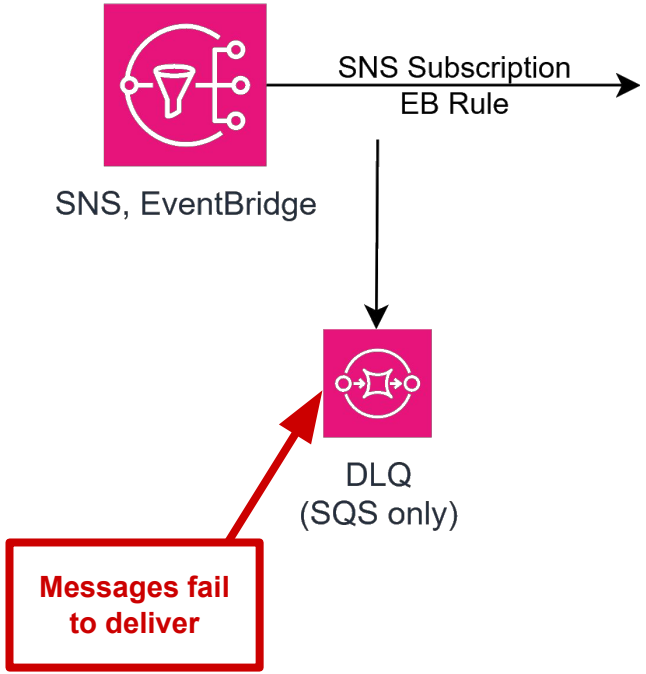


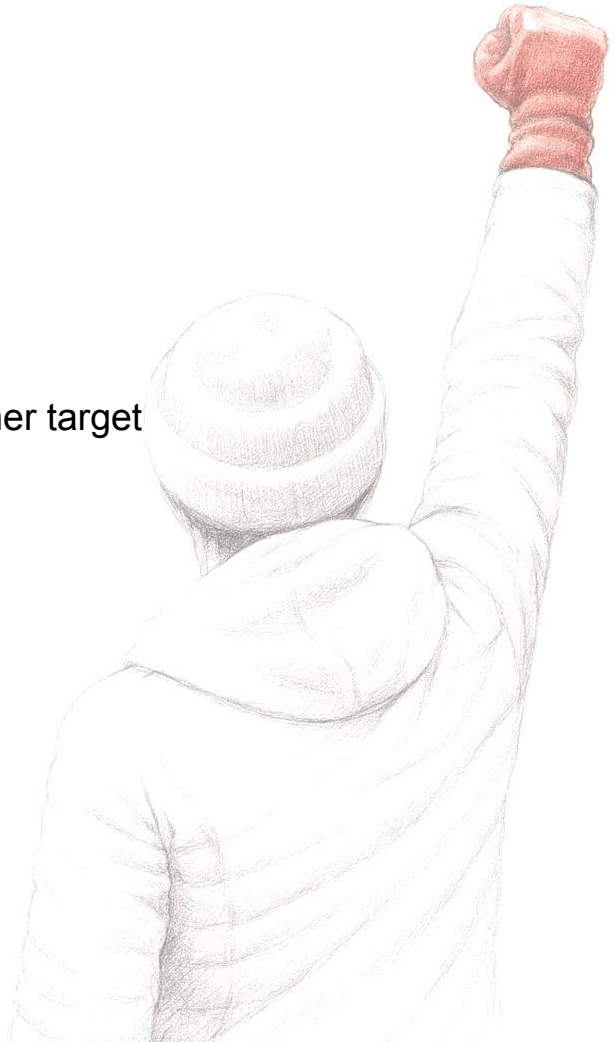


NO MORE SERVERS!

Dead-Letter Queue (DLQ)

SNS, EventBridge

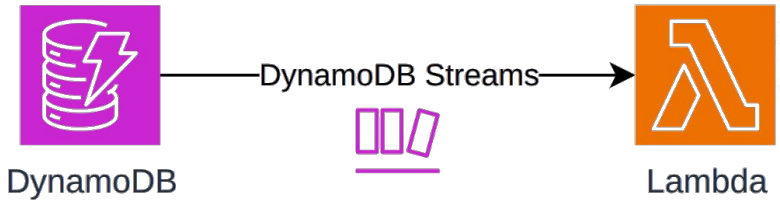


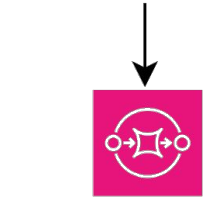


NO MORE SERVERS!

Dead-Letter Queue (DLQ)

DynamoDB Streams, Kinesis Data Streams



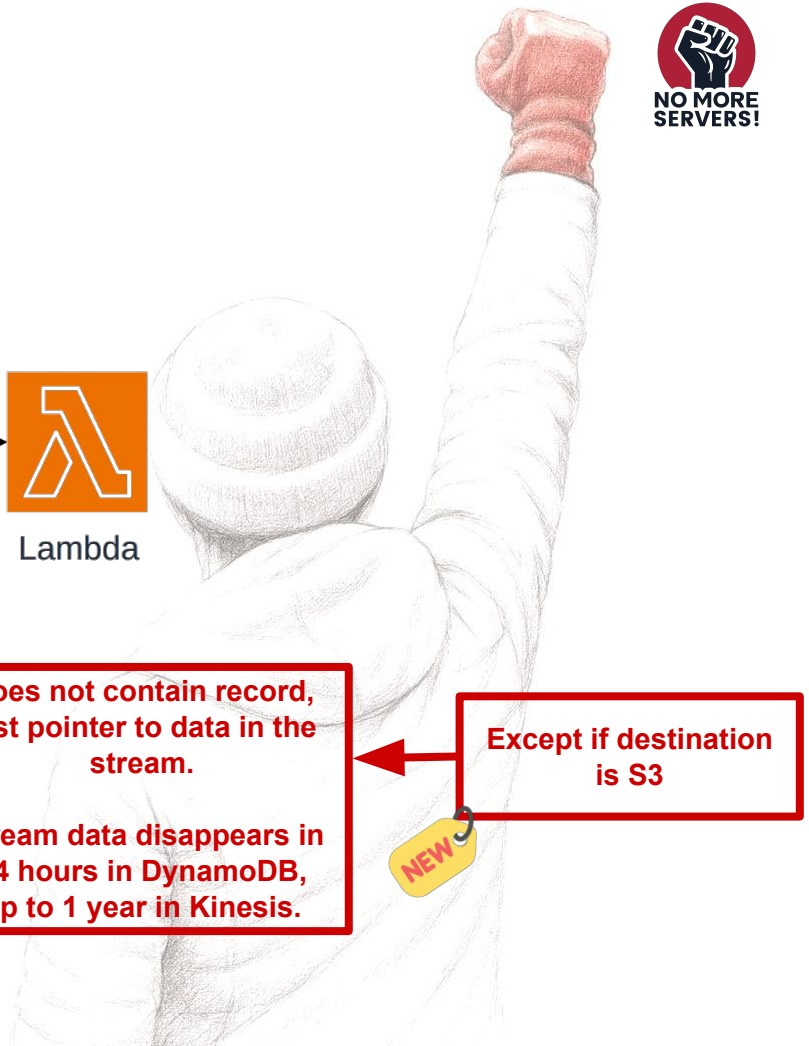


Does not contain record, just pointer to data in the stream.

Stream data disappears in 24 hours in DynamoDB, up to 1 year in Kinesis.

Except if destination is S3

NEW



Thank you!

NO MORE SERVERS!