IT Nights

Обработка озера данных на Node.js в serverless-архитектуре

Николай Матвиенко

Grid Dynamics


Well, as per usual I don't know which talk to choose...
There's a session on the future of web hosting which sounds good... or maybe the one about virtual assistants...
Don't overcomplicate it, I've found the perfect talk for us
HOW TO BUILD A CHATBOT FOR BLOCKCHAIN SMART CONTRACTS USING GRAPHQL WITH SERVERLESS APIS SUPPORTED BY A SERVICE MESH
Perfect!
CommitStrip.com

Nikolay Matvienko

Senior Software Engineer and Node.js expert at Grid Dynamics

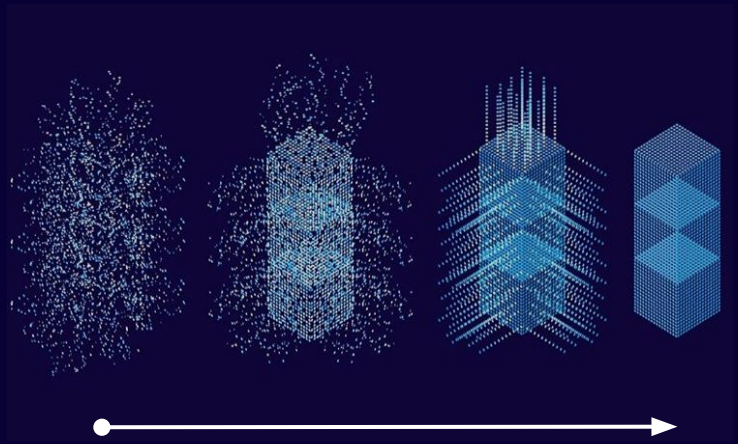
Node.js diagnostics, performance improvement consultant.

You can find me at twitter.com/**matvi3nko**

github.com/**matvi3nko**

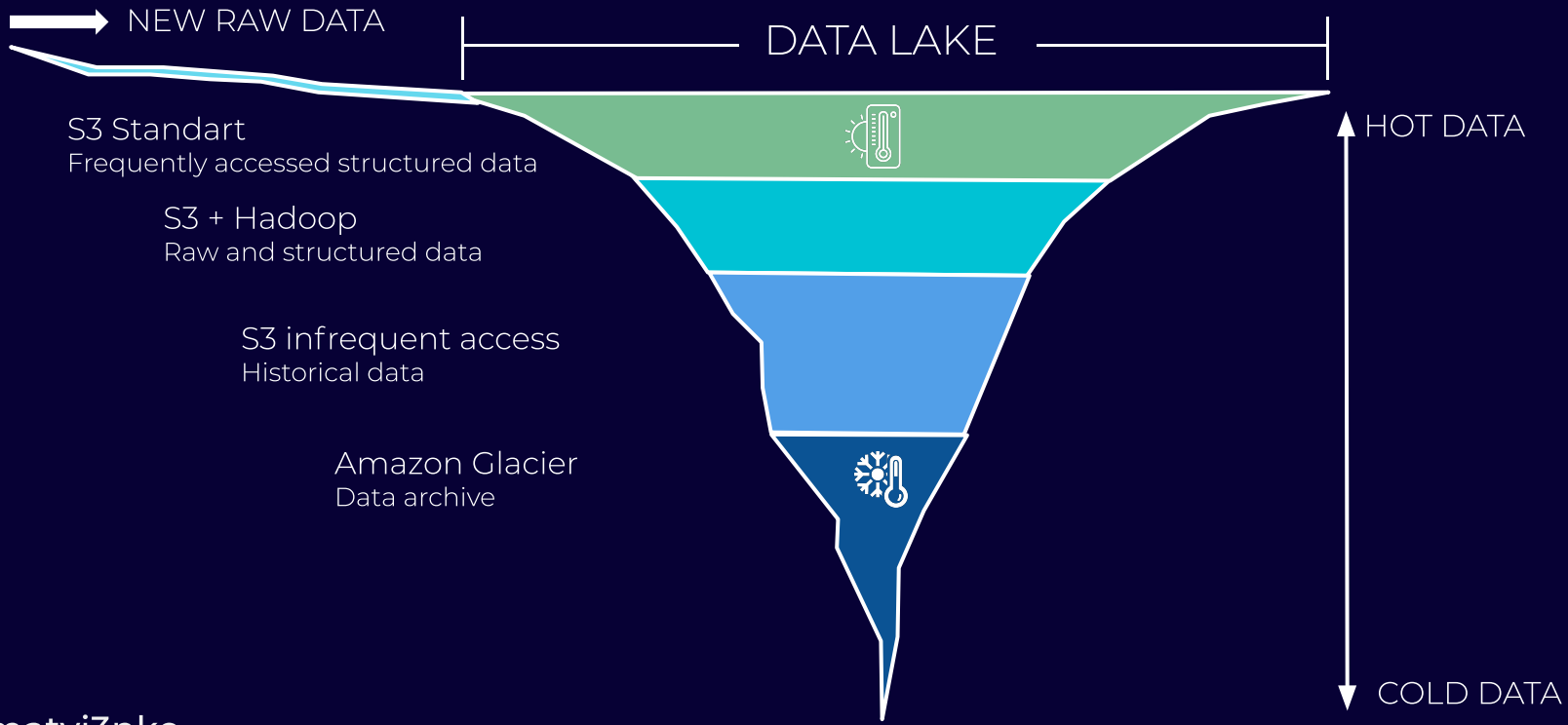

LAMBDA
DATA LAKE
NODE.JS
HOW TO
FASTER
BIG DATA
ETL / ELT
PYTHON
GO
EC2
REAL-TIME
STREAMING
SERVERLESS
BATCH

DATA LAKE



RAW DATA MODEL STRUCTURED DATA ANALYZED DATA MODEL

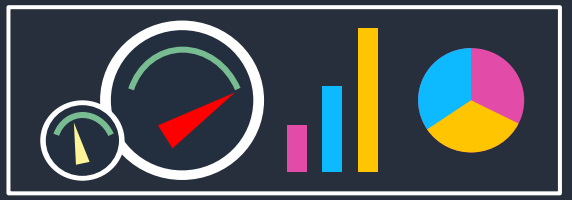
THE GOAL

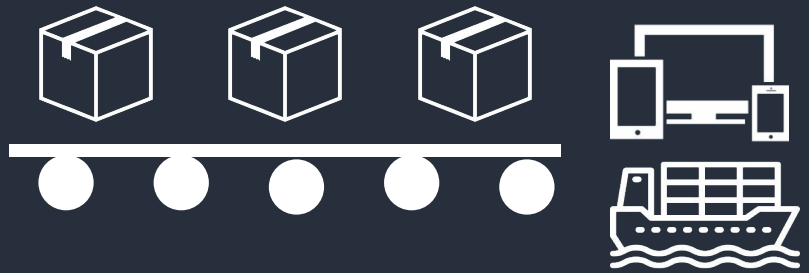

NEW RAW DATA
DATA LAKE
S3 Standart
Frequently accessed structured data
S3 + Hadoop
Raw and structured data
S3 infrequent access
Historical data
Amazon Glacier
Data archive
HOT DATA
COLD DATA

PRODUCTION

INTELLIGENCE SUPPLY CHAIN

- Hundreds GB RAW data and 60M new messages daily
- 8M UNIQUE ITEMS over the world





TECHNOLOGIES

UI

SERVERLESS BACKEND API

AWS Lambda
Node.js / GraphQL / Databases
TS

DATABASES

DATA LAKE (BIG DATA)

AWS S3
Hadoop Cloudera
pySpark

TECHNOLOGIES

UI

SERVERLESS BACKEND API

AWS Lambda
Node.js / GraphQL / Databases
TS

DATABASES

SERVERLESS DATA PROCESSING

AWS Lambda
Node.js / Databases
TS

DATA LAKE (BIG DATA)

AWS S3
Hadoop Cloudera
pySpark

SERVERLESS

"Focus on your core product instead of worrying about managing and operating servers or runtimes"

DATABASES
DATA STORES

QUEUES,
STREAMS,
NOTIFICATIONS,
EVENT BUS

FUNCTIONS
ANALYTICS
JOBS

SERVERLESS FUNCTION

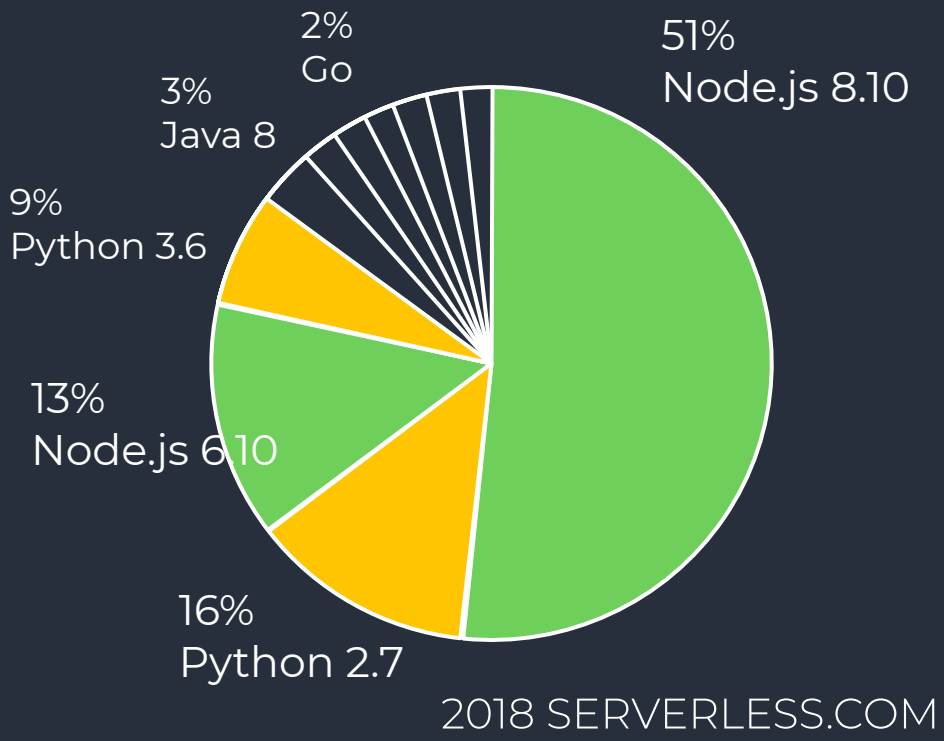
```
export const handler = async (event) => {
 const data = event.Records[0].body;

 // - TRANSFORM data
 // - WRITE to DB or
 // - PUT TO QUEUE/STREAM/TOPIC

 return 'success';
};
```

- NO INFRASTRUCTURE TO MANAGE
- TRIGGERED BY EVENTS
- HIGH SCALABLE
- STATELESS
- COST-EFFECTIVE
- CHOOSE YOUR CODE LANGUAGE

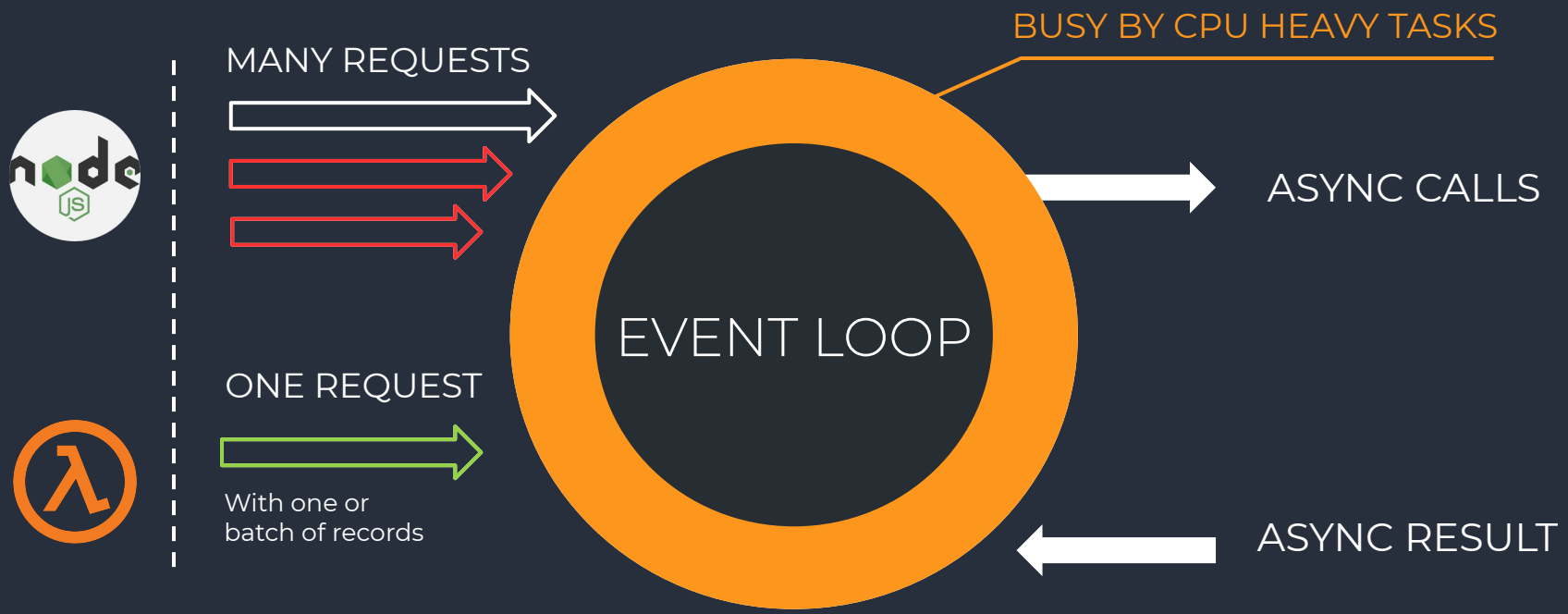
NODE.JS POPULARITY

2014 AWS LAMBDA WITH NODE.JS

- PURE ASYNCHRONOUS
- MINIMALISTIC CORE
- FAST STARTUP WITH HIGH PERFORMANCE



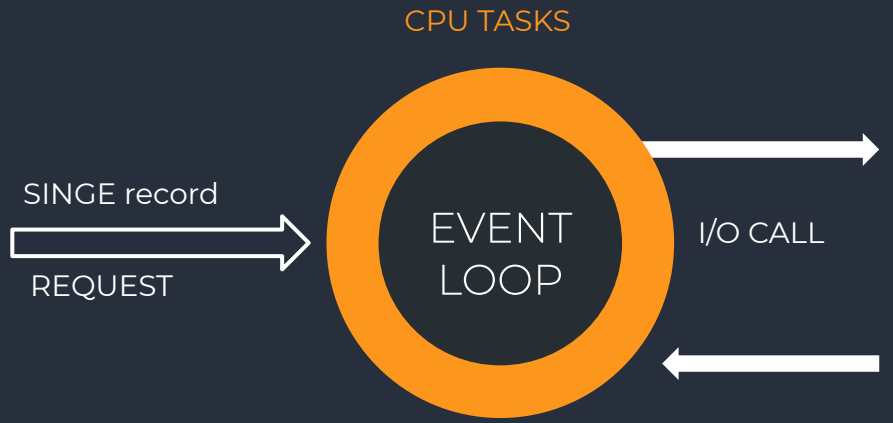
NODE.JS FUNCTION VS SERVER



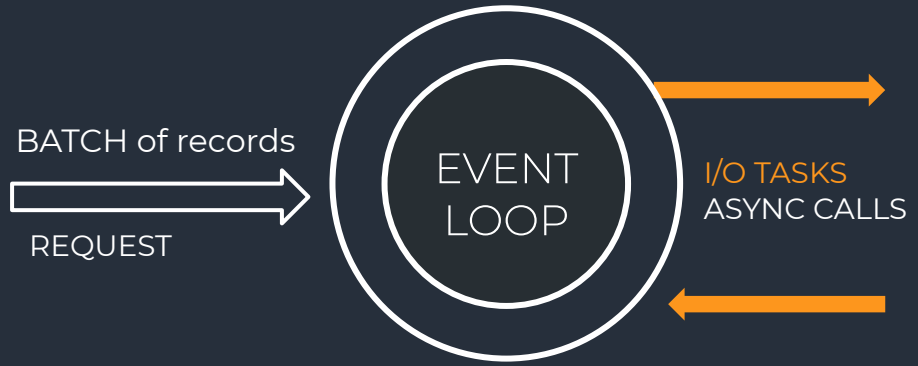
I/O – PERFORMANCE BOTTLENECK

SUMMARY

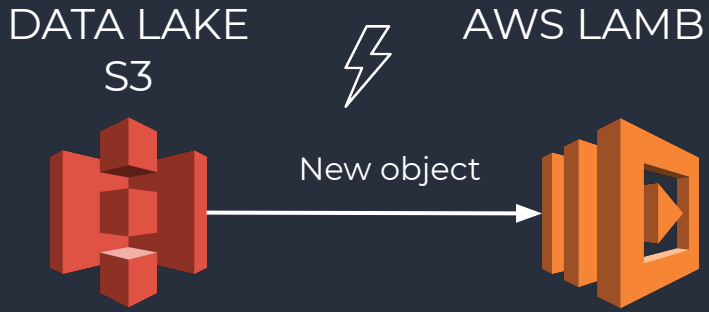
CPU HEAVY TASKS
Get request with single record



I/O INTENSIVE TASKS
Get request with batch of records (like a server)

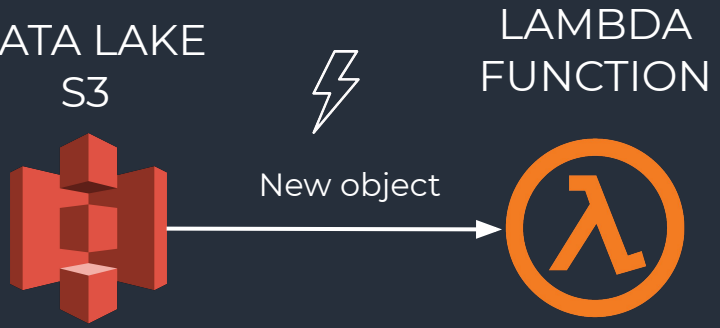


SERVERLESS COMPUTE SERVICE

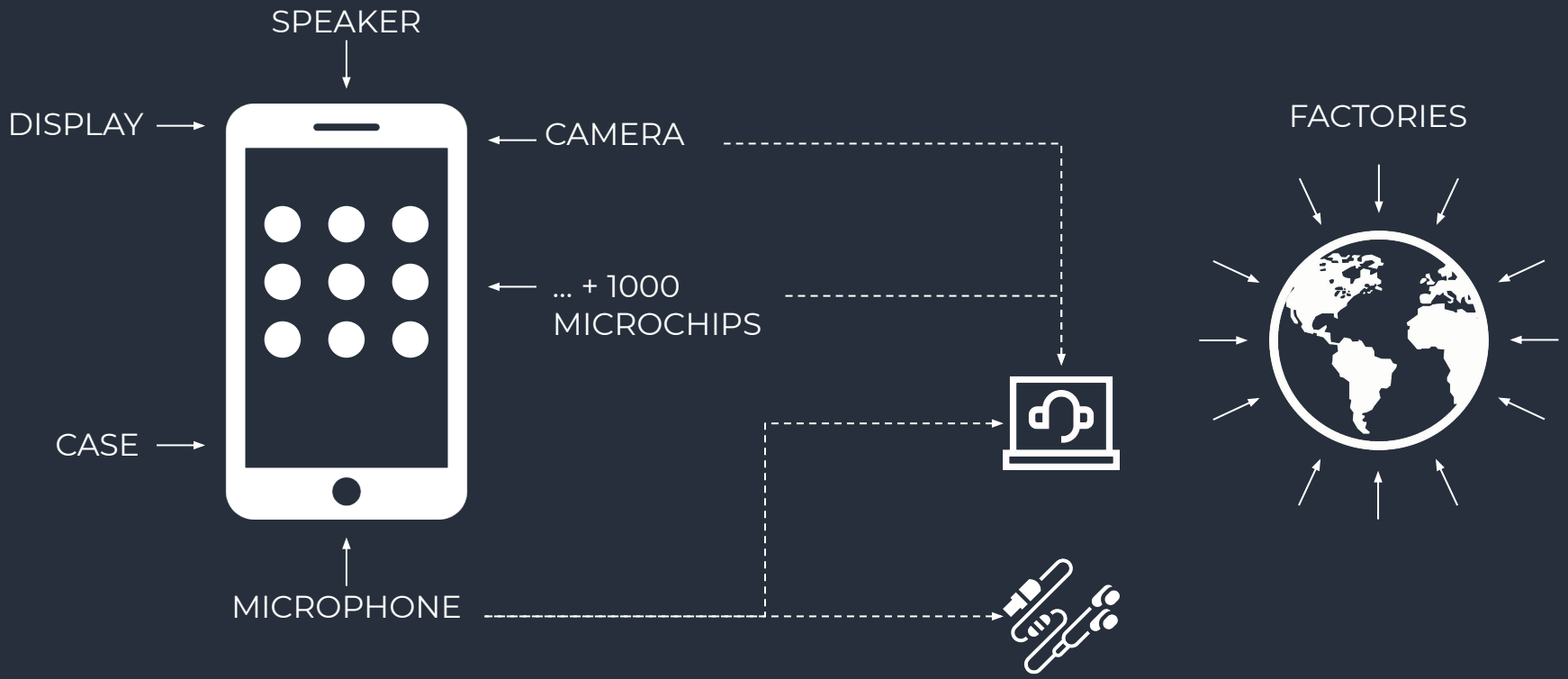


THIS IS MY ARCHITECTURE – 177+ VIDEOS
https://aws.amazon.com/ru/this-is-my-architecture/

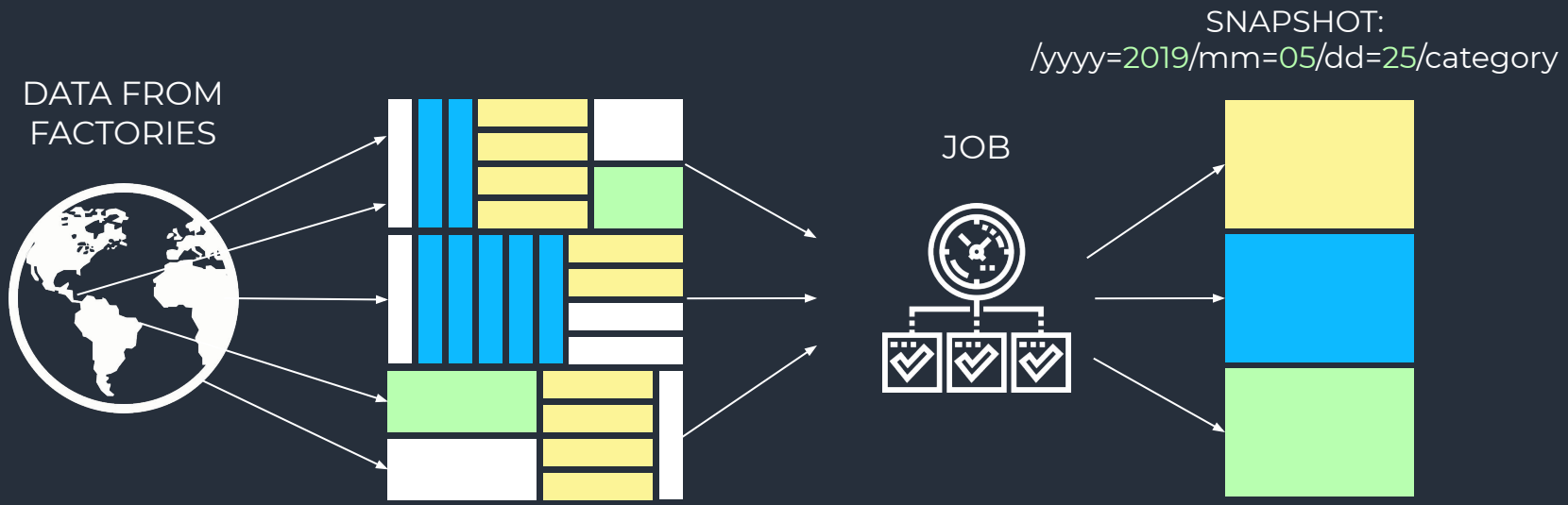
Symbol in the presentation.



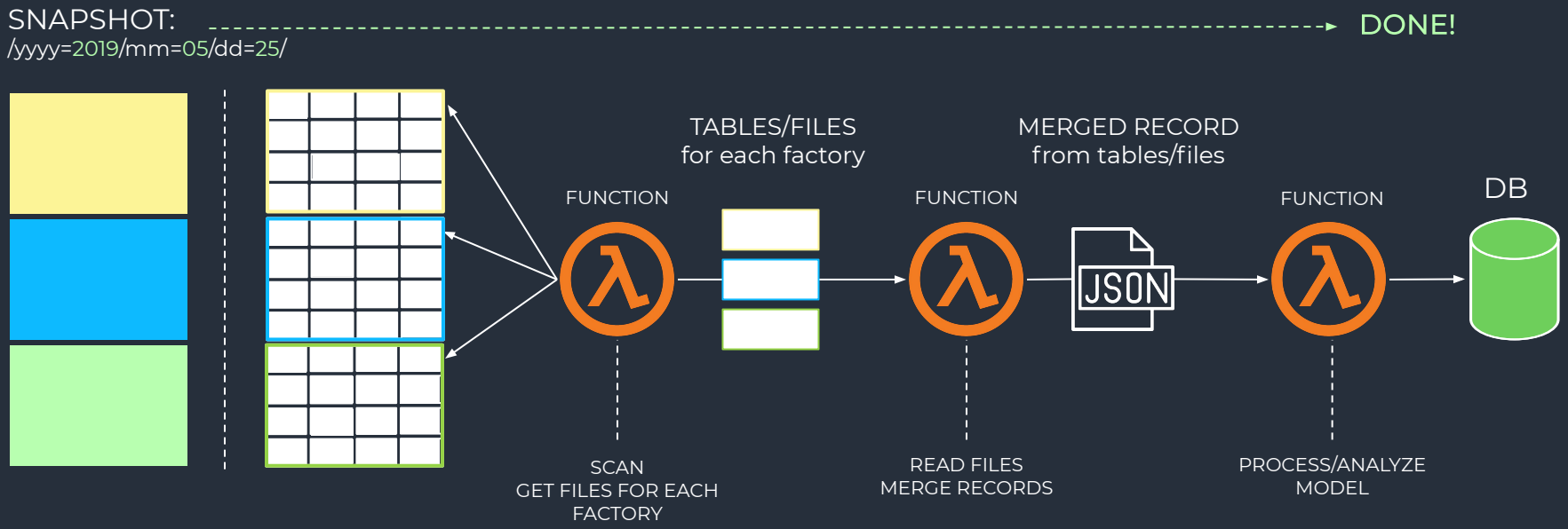
EXAMPLE


SPEAKER
DISPLAY
CAMERA
... + 1000
MICROCHIPS
CASE
MICROPHONE
FACTORIES

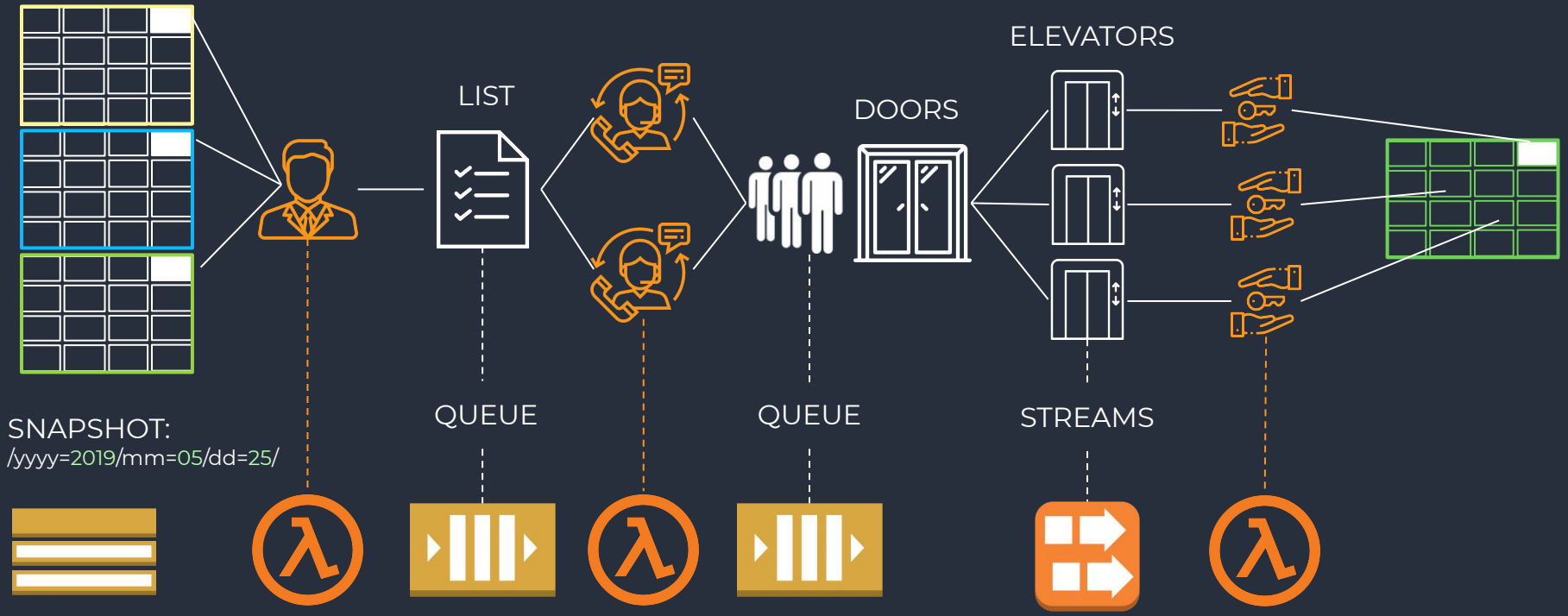
BATCH PROCESSING



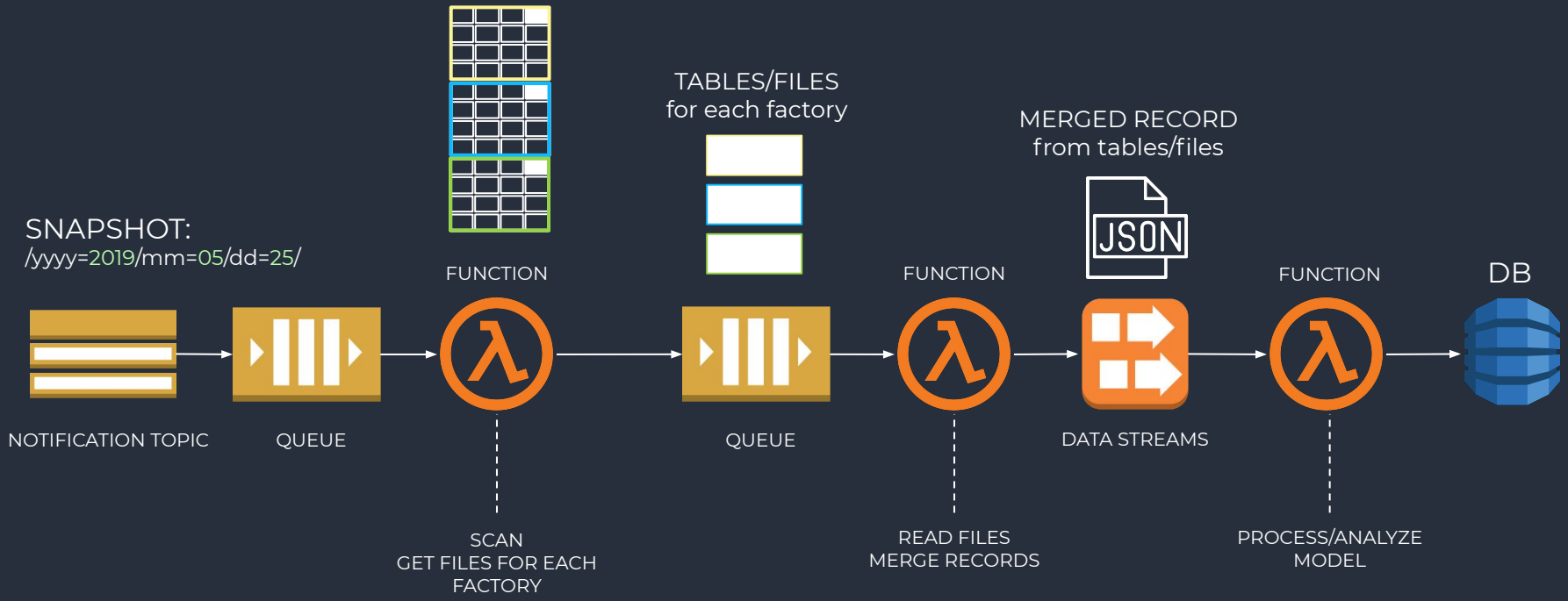
BATCH DATA PROCESSING



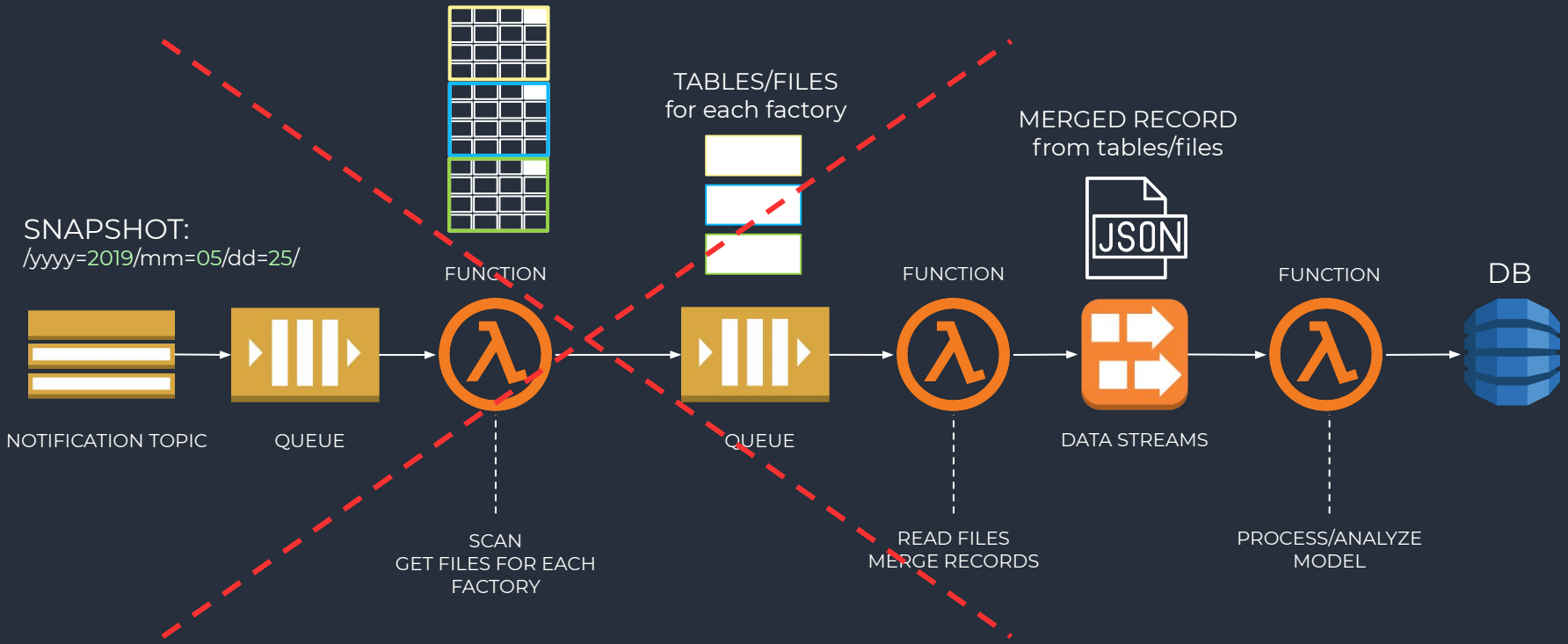
RELOCATION OF PEOPLE TO A NEW BUILDING



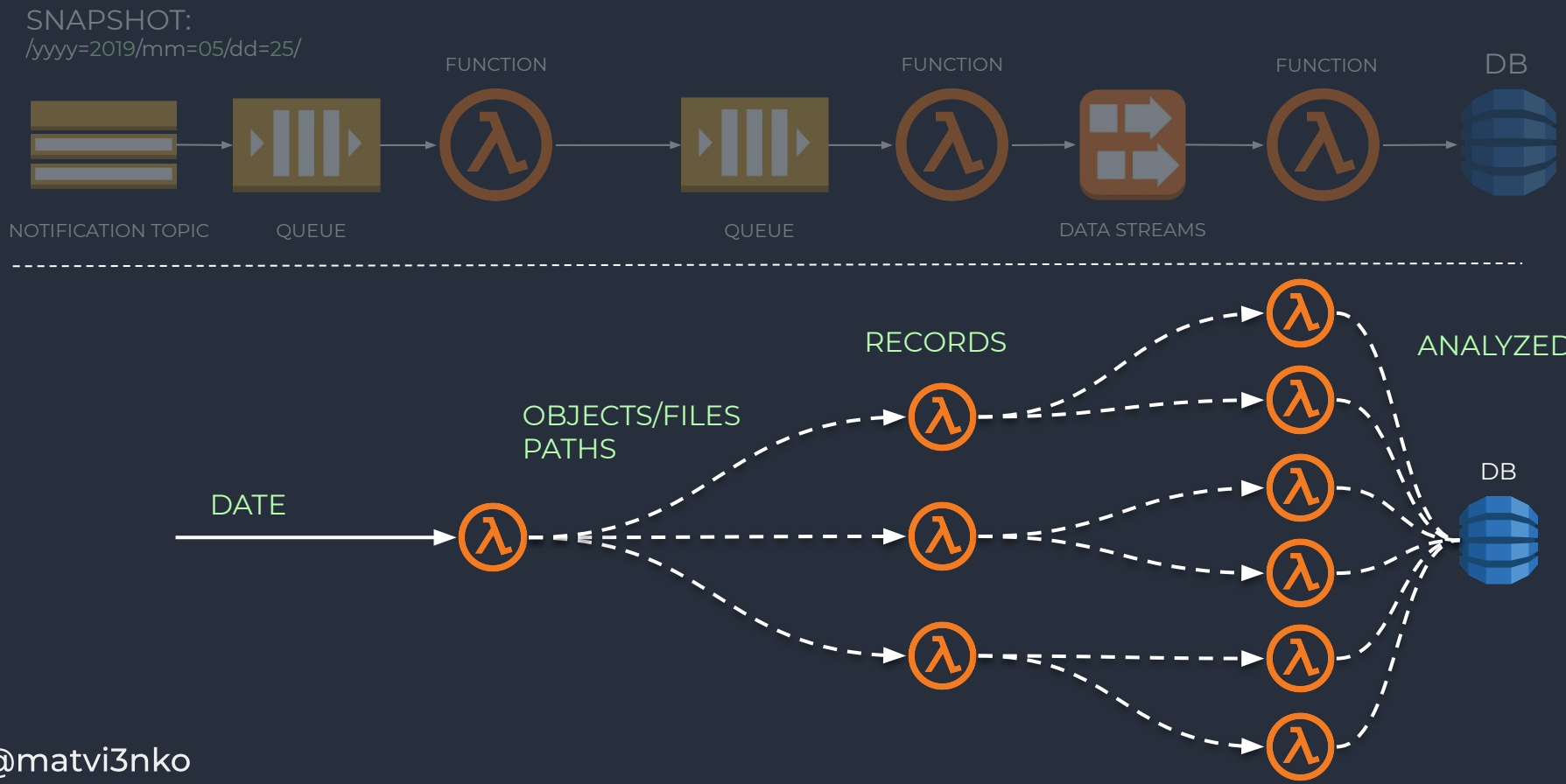
BATCH PROCESSING ARCHITECTURE



REAL-TIME PROCESSING ARCHITECTURE



SCALABILITY



CONCLUSION

1. FROM BIG DATA TO A LARGE NUMBER OF MESSAGES
2. THE MORE MESSAGES THE FUNCTION ACCEPTS, THE MORE IT NEEDS TO BE PARALLELIZED
3. USE THE QUEUE FOR MESSAGES, AND DATA STREAMS TO TRANSFER MODELS / LARGE COLLECTION
4. INCREASE THE NUMBER OF STREAM SHARDS. 1 SHARD = 1 LAMBDA FUNCTION
5. PREPARE TO STREAMING / REAL-TIME PROCESSING

PROGRESS

SOLUTION STRUCTURE AND FUNCTION

BASE ARCHITECTURE DESIGN

SERVERLESS PROJECT STRUCTURE

/transform
 – serverless.yml
 – handler.ts
/analyze
 – serverless.yml
 – handler.ts
/node_modules
serverless.yml
package.json

```
import AWS from 'as-sdk';
const s3Client = new AWS.S3({region});

export const handler = (event) => {
   const [message] = event.Records;

   return new Promise((resolve, reject) => {
     this.s3Client.selectObjectContent({ Key: message.path }, (err, data) => {
     if (err) {
        reject(err);
     }
     resolve(data);
   });
};
```

SERVERLESS PROJECT STRUCTURE

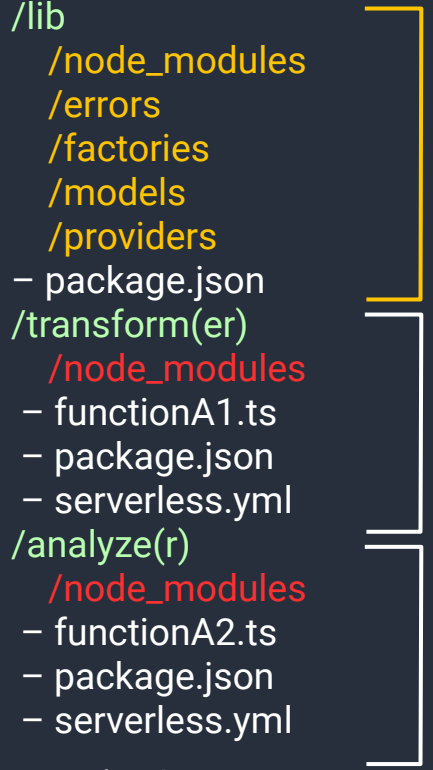
/transform
 – serverless.yml
 – handler.ts
/analyze
 – serverless.yml
 – handler.ts
/node_modules
serverless.yml
package.json

DISADVANTAGES

1. NODE_MODULES contains dependencies of all functions

   Have to control and split them in SERVERLESS.YML
2. Lack of function isolation
3. Lack of independent install / build / test
4. Becomes monolith project

MONOREPO SERVERLESS PROJECTS STRUCTURE



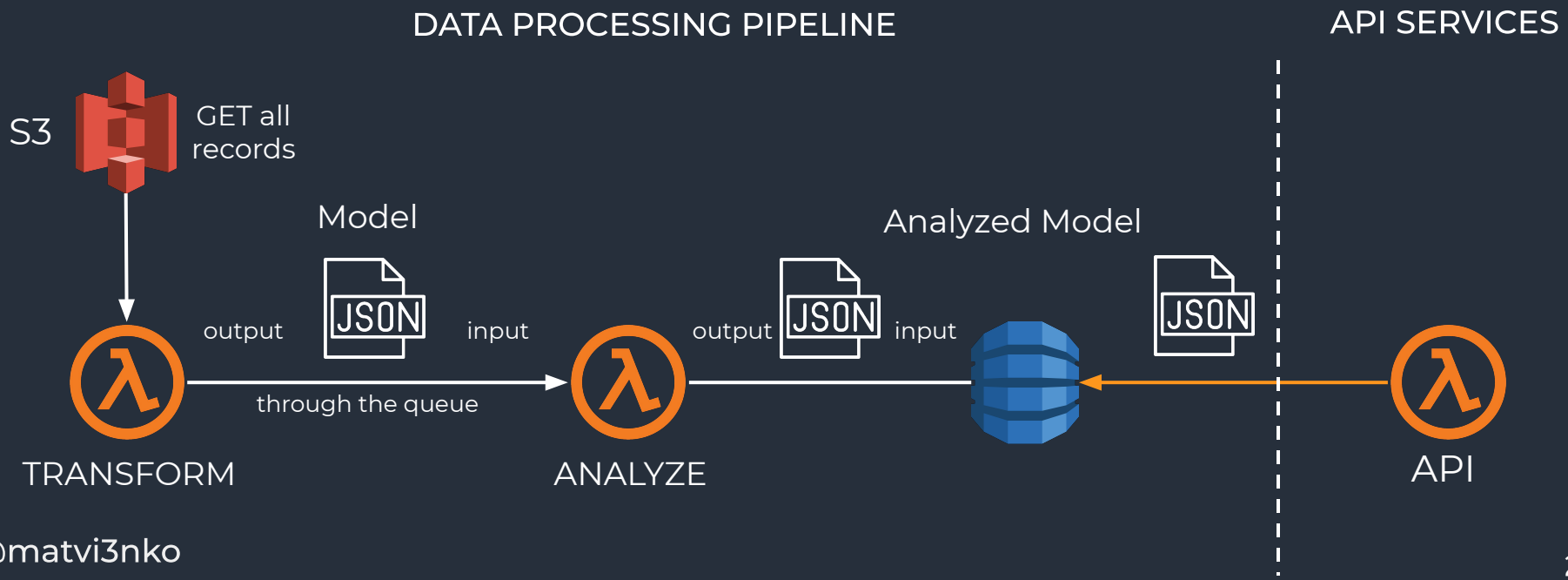
ADVANTAGES

1. LIB contains all common infrastructure, domain logic and cloud provider's SDK
2. Functions became isolated projects with flexible splitting and contains only business logic
3. LIB and PROJECTS versioning
4. NPM resolves NODE_MODULES dependencies automatically
5. Independent install / build / test / deploy / troubleshooting

@matvi3nko

MODELS REUSABILITY

```
import { Model } from '@holyjs/models'
```

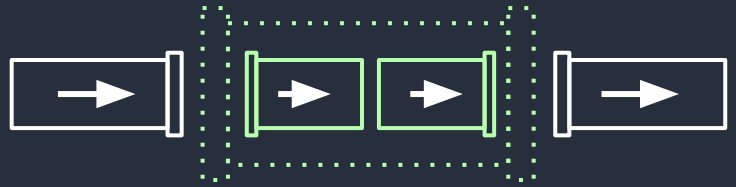


3 MAIN PRINCIPLES OF ANY FUNCTION

1. INITIALIZE ONLY ONCE
2. USE STREAMING PROCESSING
3. HANDLE ERRORS AT A HIGH LEVEL

RECOMMENDED

```
const s3 = createCsvS3Provider(config.region);
const streams = createKinesisProvider(config.region);
const service = new SomeService();

const handler = async (event) => {
 const source = s3.getObjects(event.Records[0].body.path)
  .pipe(flatMap(([a, b, c]) => from(service.transform(a, b, c)))
  .pipe(flatMap(model => service.compress(model)))
  .pipe(bufferCount(config.batchSize))
  .pipe(flatMap(batch => streams.put(batch)));

 return new Promise((resolve, reject) => {
  source.subscribe(() => { /* handle */},
    err => {
      err instanceof InfrastructureError && reject(err);
      err instanceof DomainError && reject(err);
    },  resolve);});};
```



NOT RECOMMENDED

```
const handler = async (event) => {
 const s3 = createCsvS3Provider(config.region);
 const streams = createKinesisProvider(config.region);
 const service = new SomeService();

 const objects = await s3.getObjects(event.Records[0].body.path);
 const models = await service.transform(objects);

 return Promise.all(models.map(model => {
   const compressedModel = await service.compress(model);
   return streams.put(compressedModel);
 });
};
```

4 COMPONENTS OF THE FUNCTION

1. INCOMING DATA
2. TYPE OF TRANSFORMATION
3. DESTINATION
4. MAIN ERROR HANDLER

PROGRESS

ERROR HANDLING

SOLUTION STRUCTURE AND FUNCTION

BASE ARCHITECTURE DESIGN

TRACK LOG MESSAGES THAT ARE HIDDEN ERRORS

1. Request XX-YY: “Process exited before completing request”
2. Function completed on its timeout (up to limit)

HOW TO HANDLE

1. FUNCTION HANGS

   Don't do that: `context.callbackWaitsForEmptyEventLoop = false;`

   or close Sequelize connection to fix that

   Use callback cb(), rewrite to async/await (Promises)

2. FUNCTION DOES NOT PERFORM PART OF THE LOGIC

   You added async or return value.

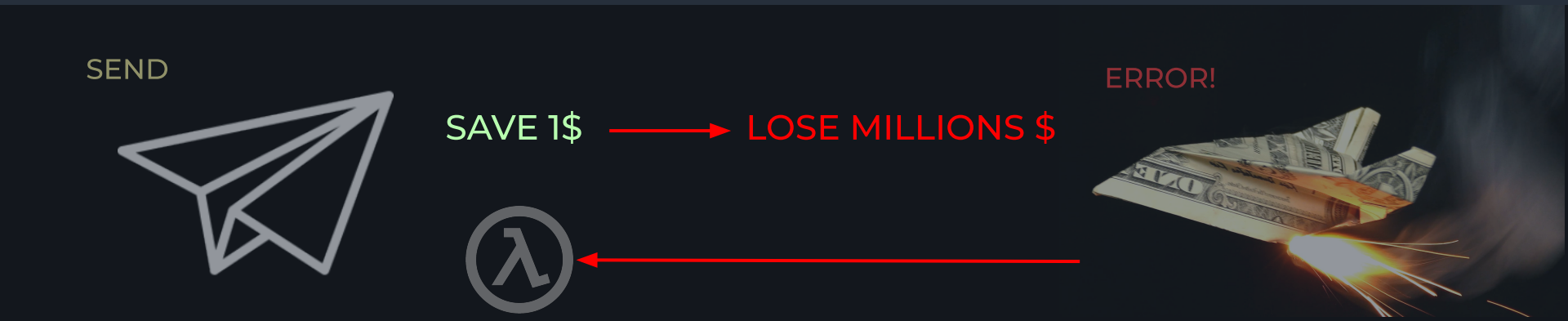
   Find missed await / return Promise

WAIT FOR RESPONSES FROM THE SERVICES

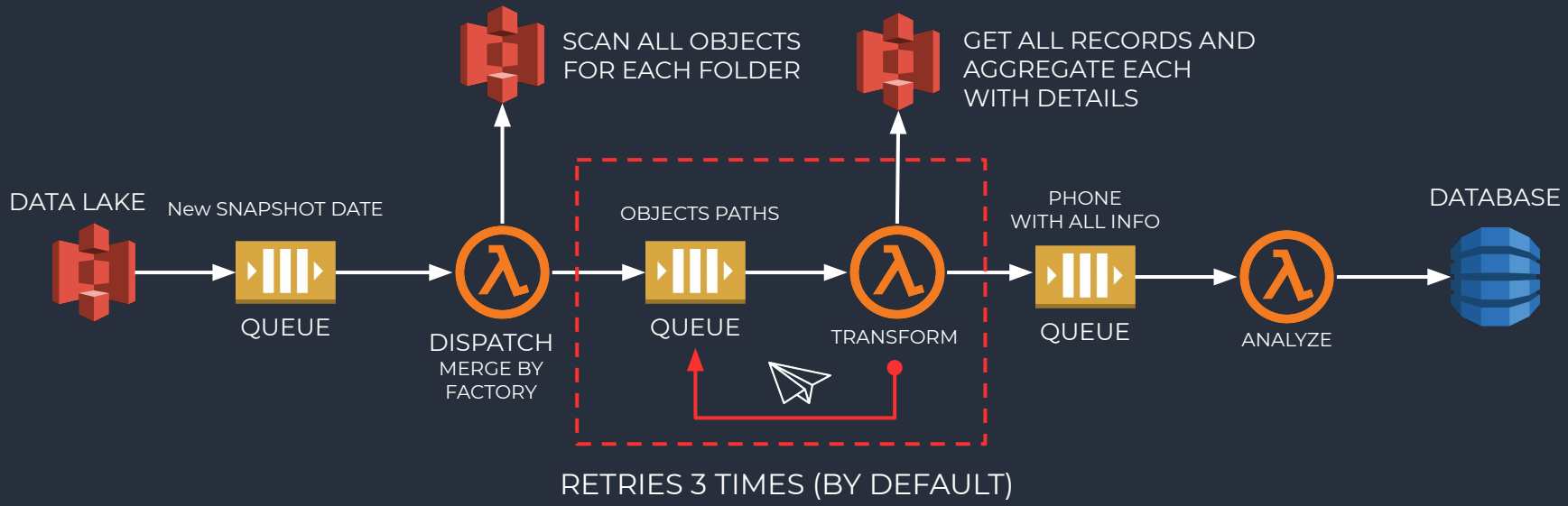
```
// CODE
SEND(MESSAGE);
// CODE
```

```
// CODE
RETURN SEND(MESSAGE);
// CODE
```

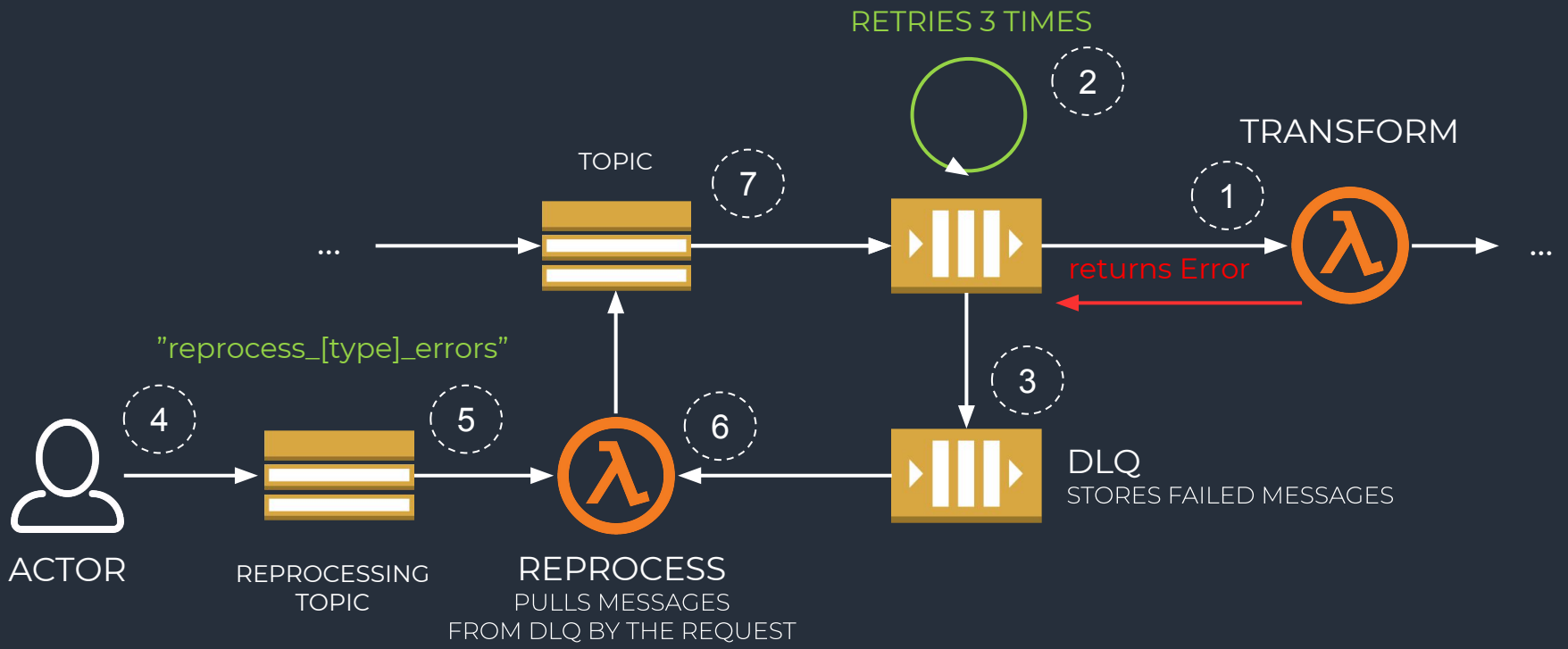
```
// CODE
AWAIT SEND(MESSAGE);
// CODE
```



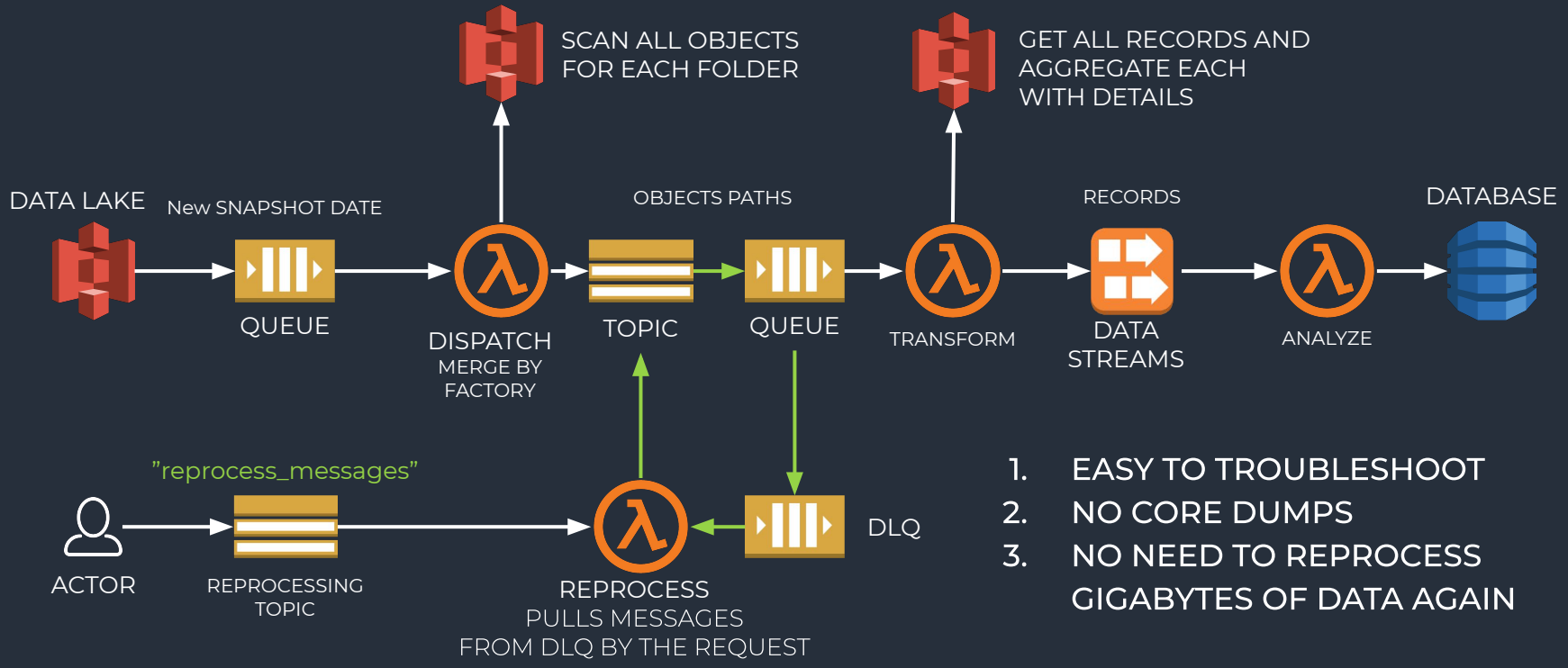
RETRY STRATEGY



DEAD LETTER QUEUE

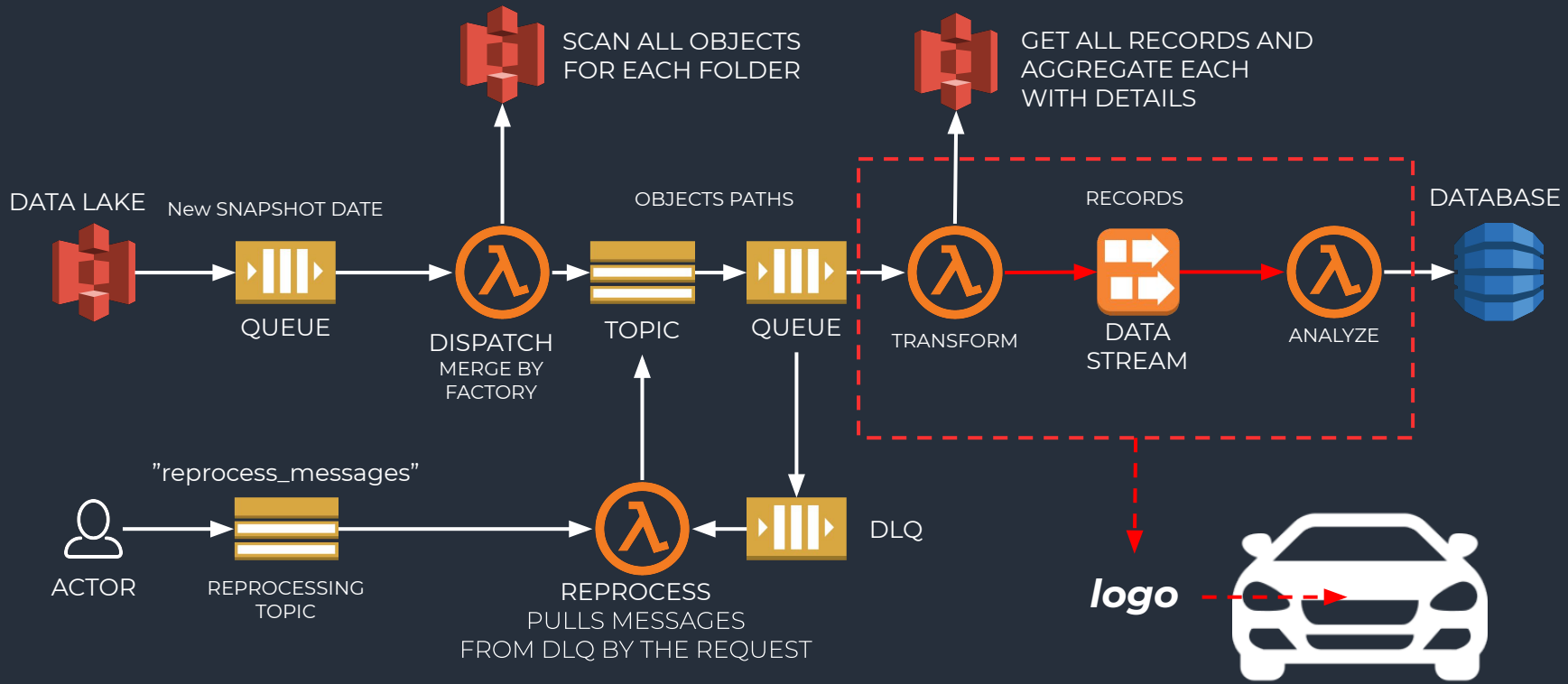


DLQ FOR THE QUEUE



1. EASY TO TROUBLESHOOT
2. NO CORE DUMPS
3. NO NEED TO REPROCESS GIGABYTES OF DATA AGAIN

KINESIS ERROR HANDLING



ERROR HANDLING STRATEGY

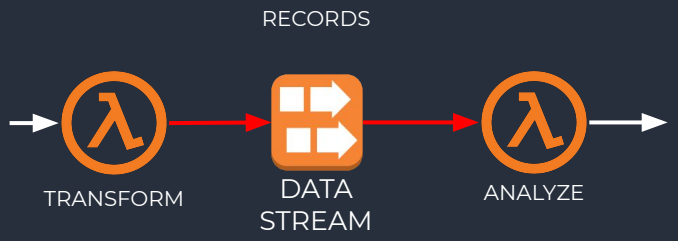
1. PUT TO STREAM (TRANSFORM)
   a. Handle partially successful response

      `FailedRecordCount : Number`

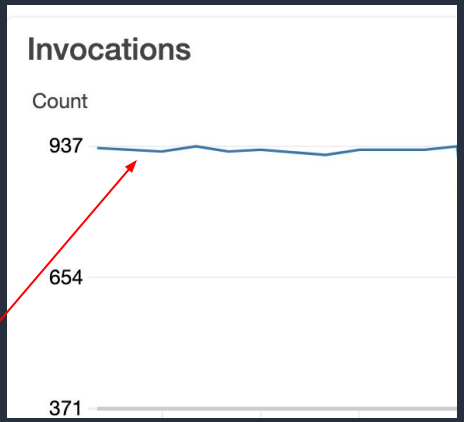
2. READ FROM (ANALYZE)
   a. Reduce retry times

      `customBackoff:  (retryCount) => {...}`

   b. Use Dead Letter Queue
   c. Handle duplicate Records



UP TO 7 days each 100 ms



EXACTLY ONCE PROCESSING

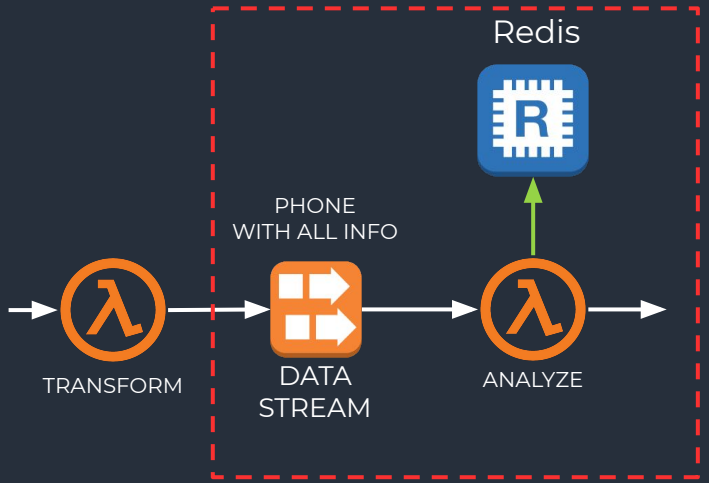
1. REDUCE THE RISK OF FAILURE

   ```
   const response = await putToStream(record);
   // do something with response –> RISK
   ```

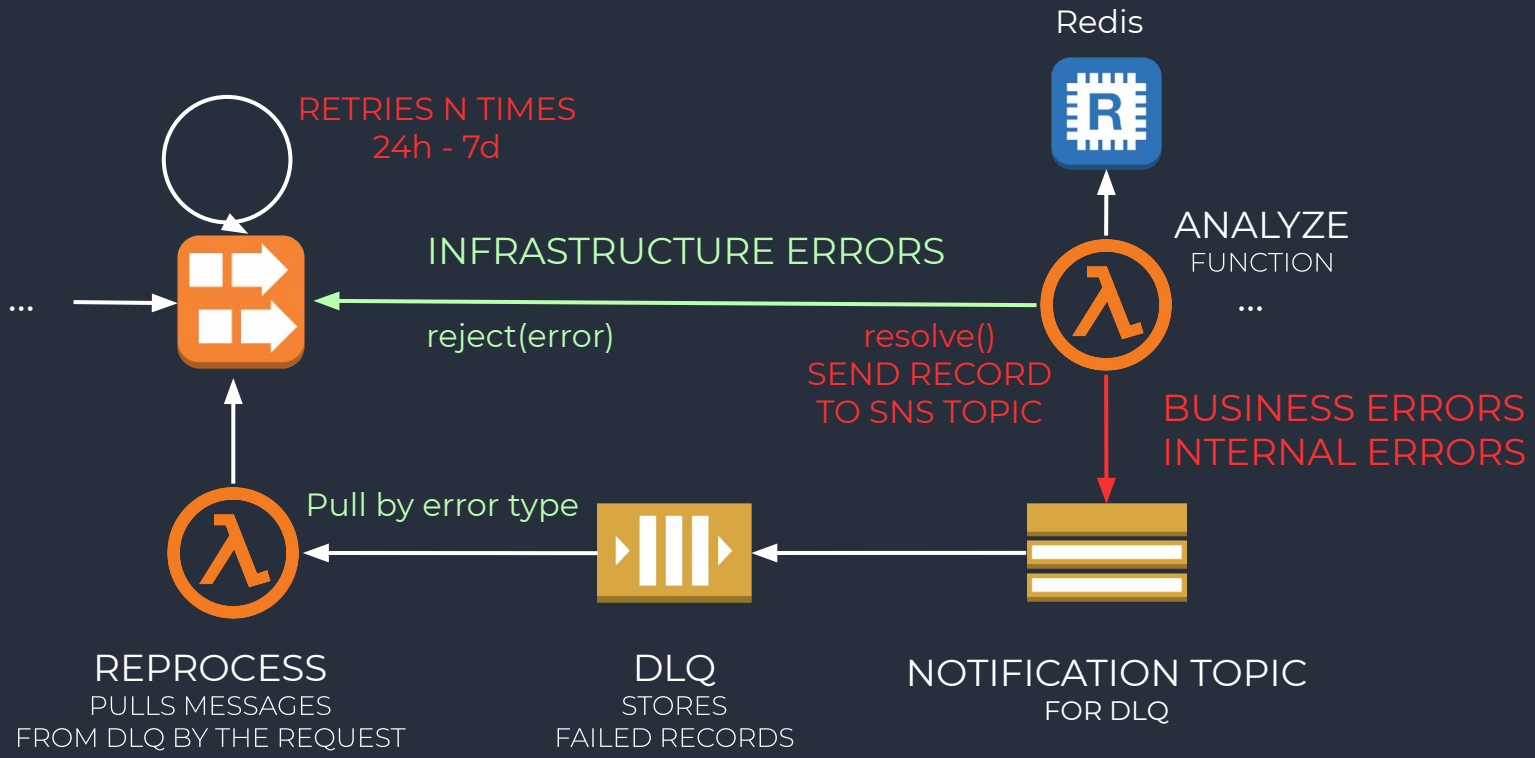
2. USE REDIS CACHE TO STORE KEYS OF RECORDS

   key: [date]-[shard-id]-[sequence-number]

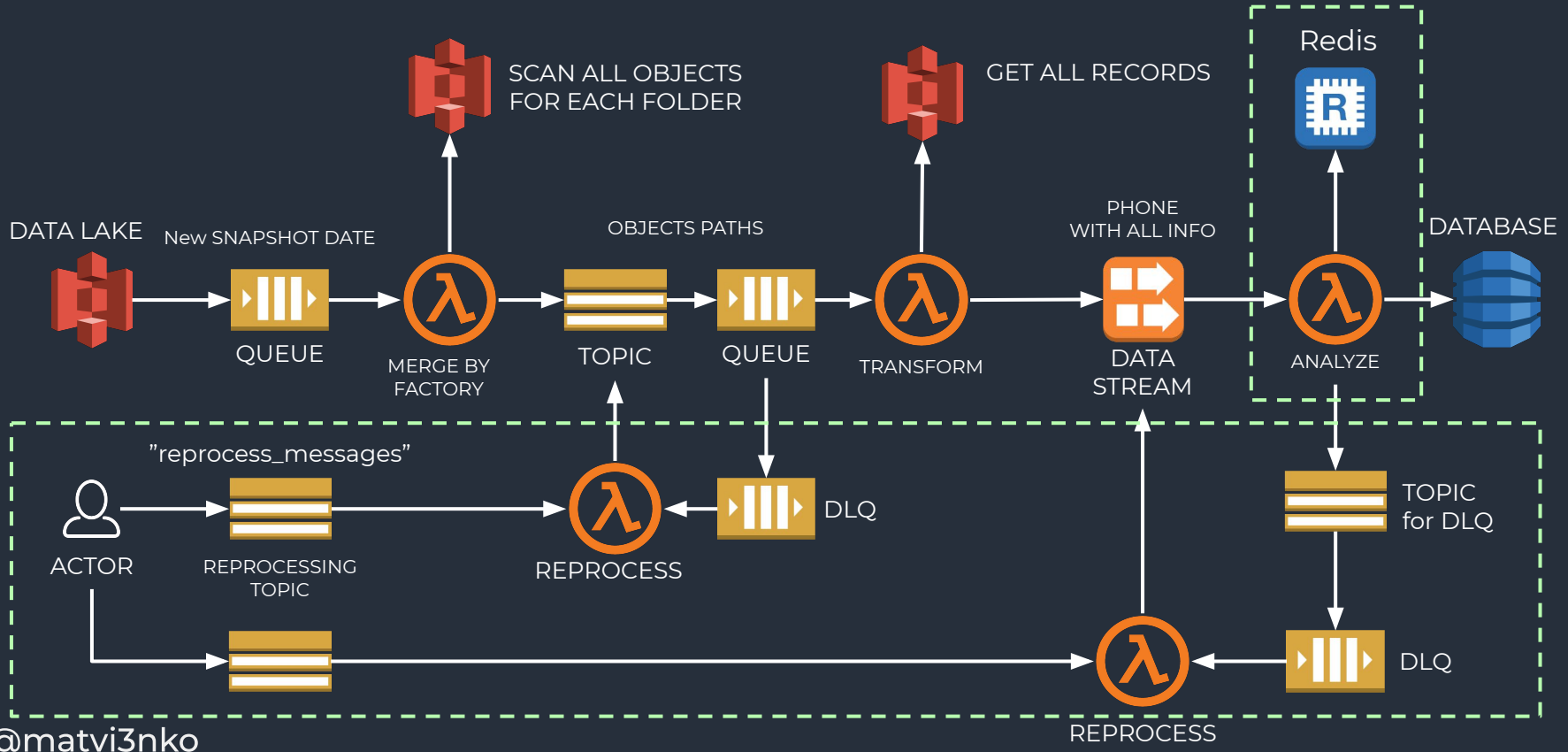
3. FILTER DUPLICATES



KINESIS ERROR HANDLING



ARCHITECTURE WITH REPROCESSING BLOCK



CONCLUSION

1. CREATE CUSTOM TYPES OF ERRORS
2. REPROCESS ONLY FAILED MESSAGE AND NOT GIGA/TERABYTES OF DATA
3. DON’T LOSE MESSAGES
4. USE DLQ WITH ERROR FILTERING
5. PROCESS EXACTLY ONCE

PROGRESS

READ AND PROCESS DATA

ERROR HANDLING

SOLUTION STRUCTURE AND FUNCTION

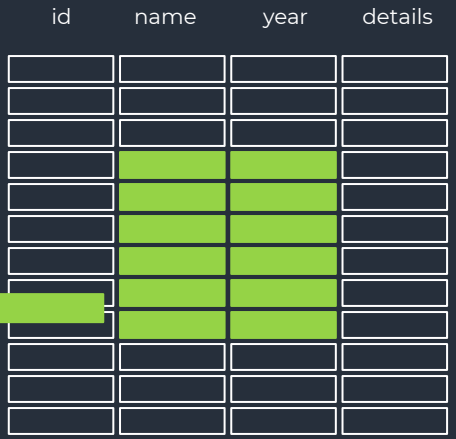
BASE ARCHITECTURE DESIGN

AWS S3: SIMPLE STORAGE SERVICE

1. S3 SELECT REQUEST

```
const params = {
  Bucket: 'bucket_name',
  Key: 'key_name',
  ExpressionType: 'SQL',
  Expression: `
    SELECT s.name, s.year
    FROM S3Object s
    WHERE s.name = 'HolyJS'
  `,
},
 OutputSerialization: {
  JSON: { RecordDelimiter: '\n' },
 };
```

2. QUERY RESULT



JSON | CSV | Parquet

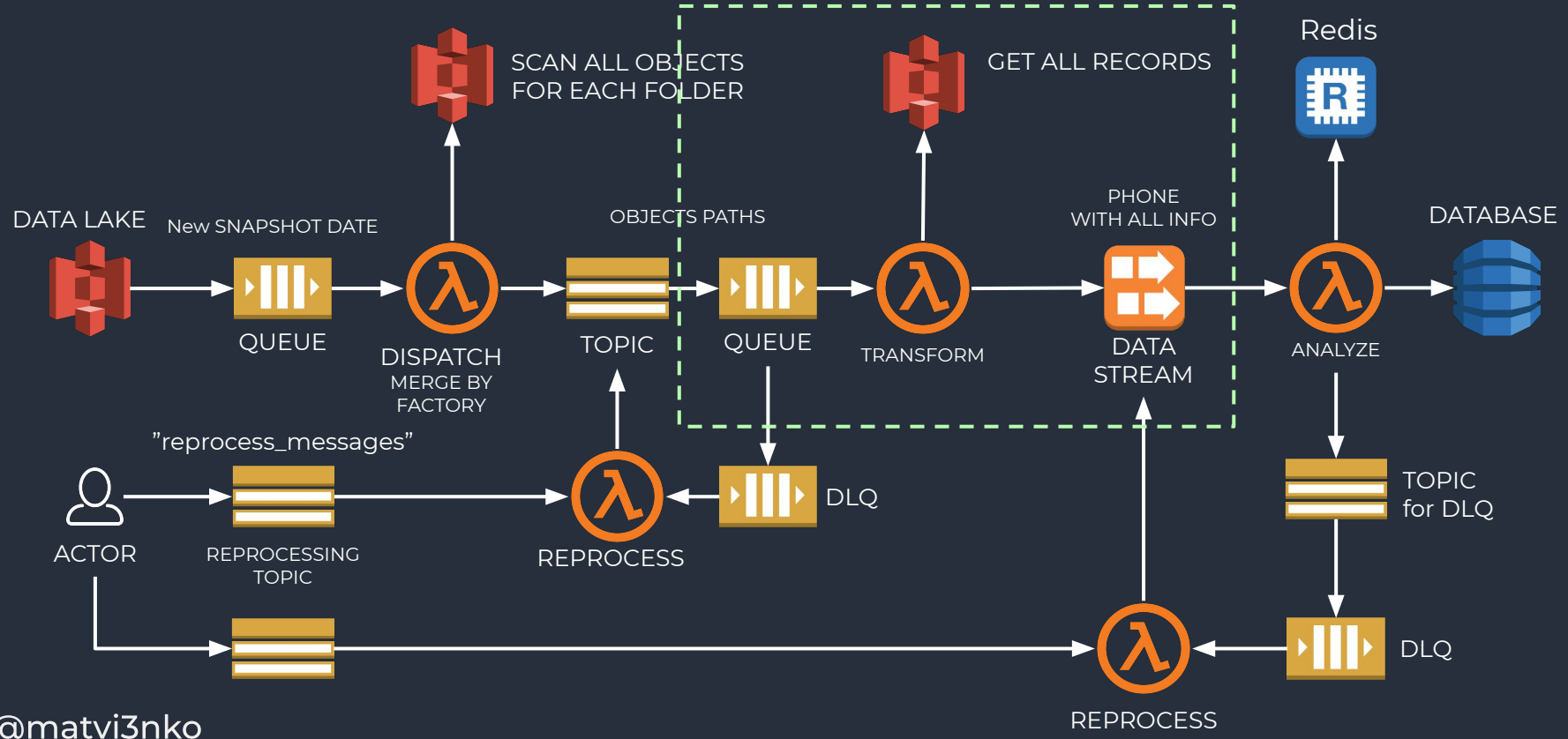
UP TO **4x FASTER**

S3: CSV VS PARQUET

Dataset	Size on Amazon S3	Query Run time	Data Scanned	Cost
Data stored as CSV files	1 TB	236 seconds	1.15 TB	$5.75
Data stored in Apache Parquet format*	130 GB	6.78 seconds	2.51 GB	$0.01
Savings / Speedup	87% less with Parquet	34x faster	99% less data scanned	99.7% savings

https://dzone.com/articles/how-to-be-a-hero-with-powerful-parquet-google-and

EXTRACTING BIG OBJECTS/FILES



DATA FORMATS

S3 OBJECT PATH:

s3/buckets/bucket-name/entity/yyyy=2019/mm=05/dd=25/partition-by-category/key.parquet

RAW OR COMPRESSED?

80MB
PARQUET
BINARY

1.4GB
CSV
TEXT. ONLY VALUES

~X20 bigger size

6.5GB
JSON
TEXT. FIELDS + VALUES

~X80 bigger size
75% - fields names

JSON

CHUNK 1 – 64KB

```
{
  sourceId: '1',
  description: 'device1',
  createdAt: "2019-02-01 00:00:00",
  someValue: "17",
  someSpecificTotalAmount: "13.50",
}, {
  sourc
```

CHUNK 2 – 64KB

```
      eId:  '2',
  description: 'device2',
  createdAt: "2019-02-01 00:00:00",
  someValue: "18",
  someSpecificTotalAmount: "14.64",
},
```

CSV

CHUNK 1 – 64KB

```
`"1","device1","2019-02-01 00:00:00","17","13.50"\n
"2","device2","2019-02-01 00:00:00","18","14.64"\n
"3","device3","2019-02-01 00:00:00","19","15.11"\n`
"4","device4","2019-02-"` ...
```

CHUNK 2 – 64KB

```
...`"01 00:00:00","20","16.43\n"
"5", "device5","2019-02-01 00:00:00","19","15.09\n"
"6", "device5","2019-02-01 00:00:00","19","15.09\n"
"7", "device5","2019-02-01 00:00:00","19","15.09\n"
```

PARQUET

```
01010110011100011011101101
00111000110111011011001110
01101110110110011100011011
01
```

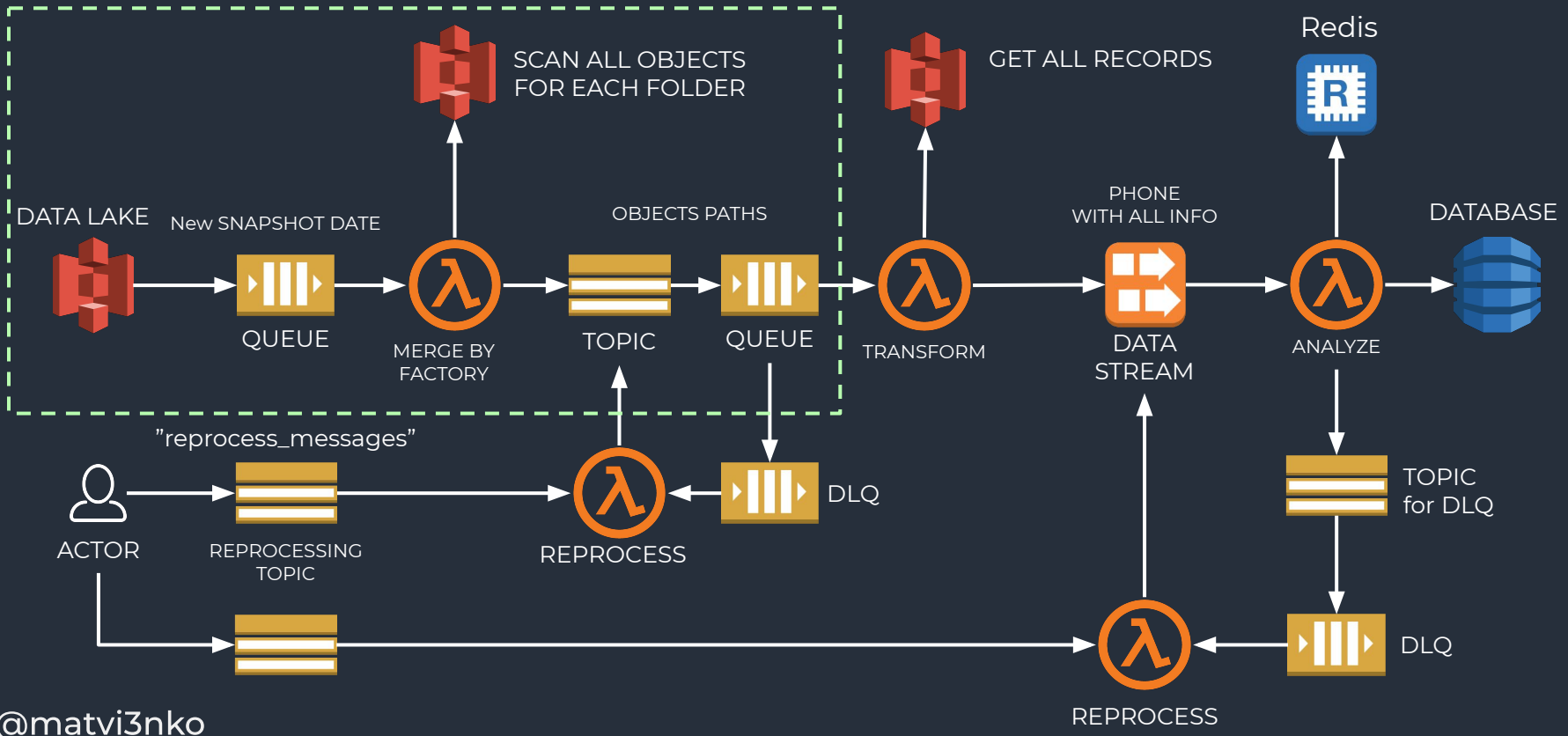
```
<Buffer 61 61 3a 20 34 2c 20 62
3a 20 35 2c 20 63 3a 20 34 2c 20
20 31 30 30 30>
```

+ DECODING IN STREAM

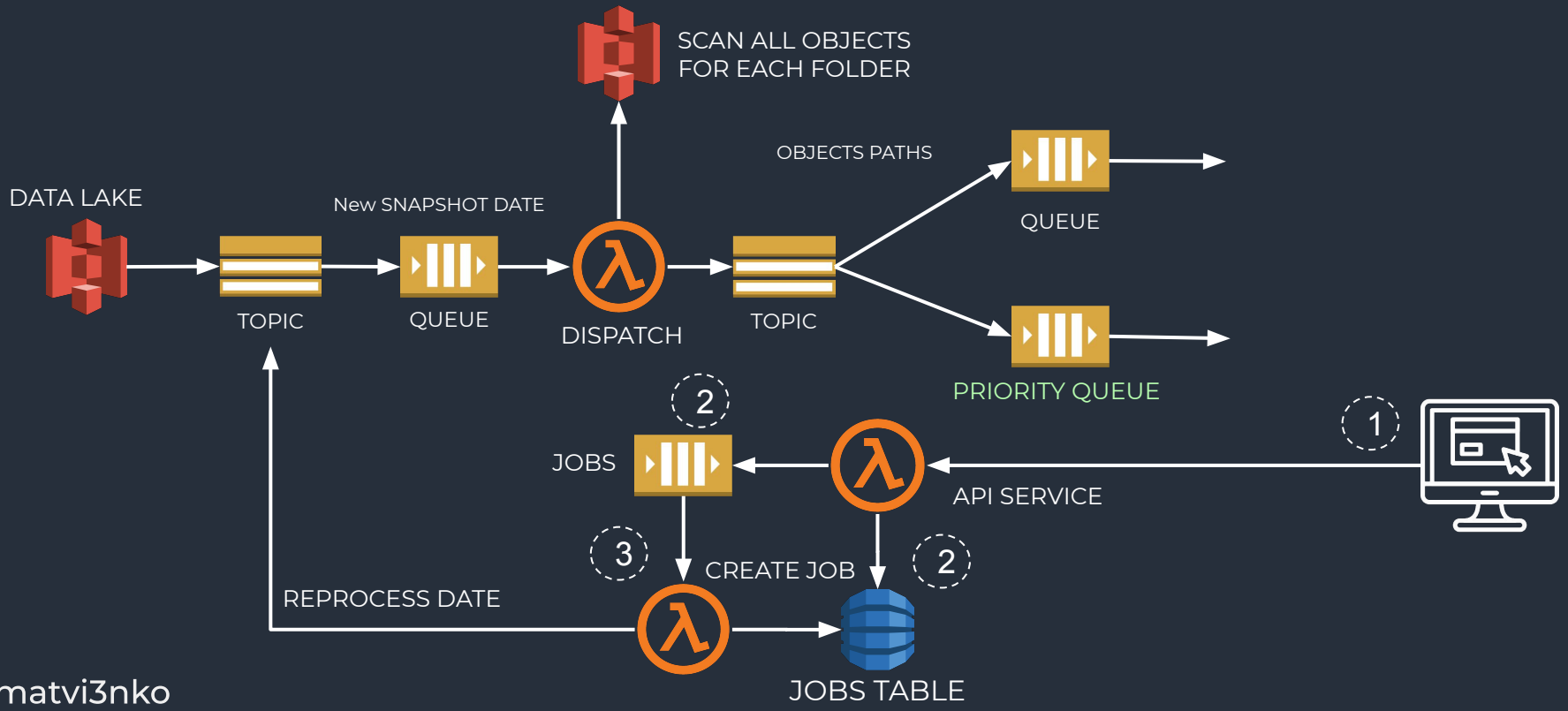
CONCLUSION

1. DON’T USE JSON FOR READING/PROCESSING
    a. Big size
    b. Slow parsing
2. USE PARQUET (or CSV in some cases)
    a. Parquet decoding Increases GC time (5 - 15% of time)
3. BUT DON’T BUFFER RESPONSE, WORK WITH STREAMS
4. GZIP RECORDS BEFORE PUT TO KINESIS STREAM
5. INCREASE MEMORY (CPU AUTO-LY) TO WORK WITH BIG OBJECTS/FILES

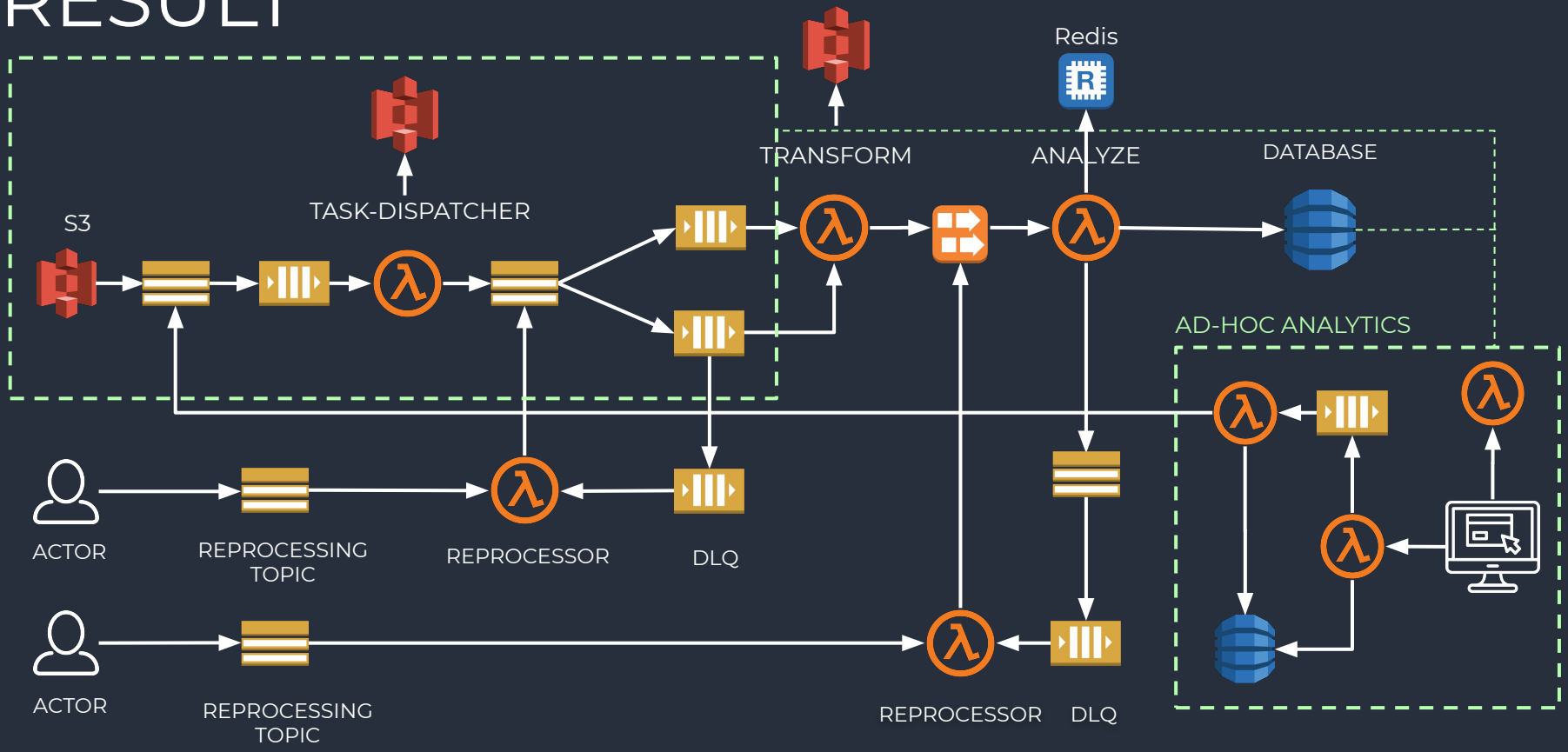
AHEAD OF THE QUEUE



ON-DEMAND REPROCESSING

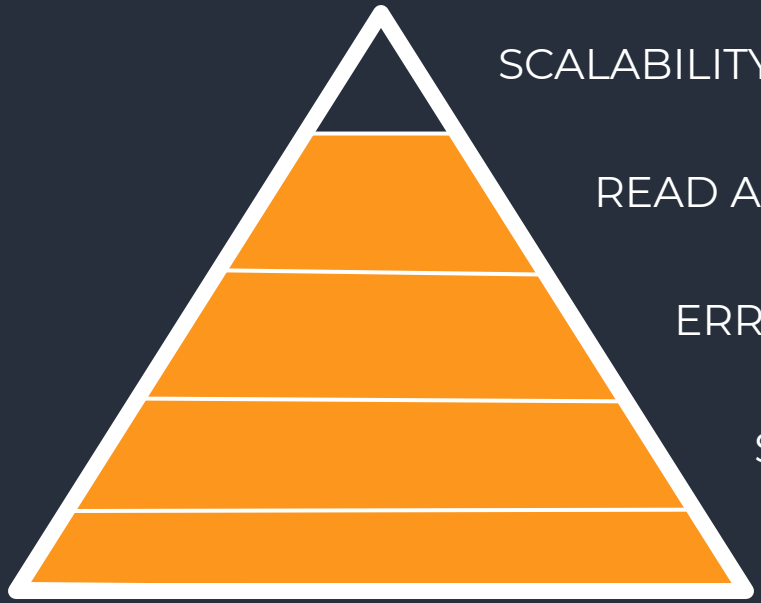


RESULT

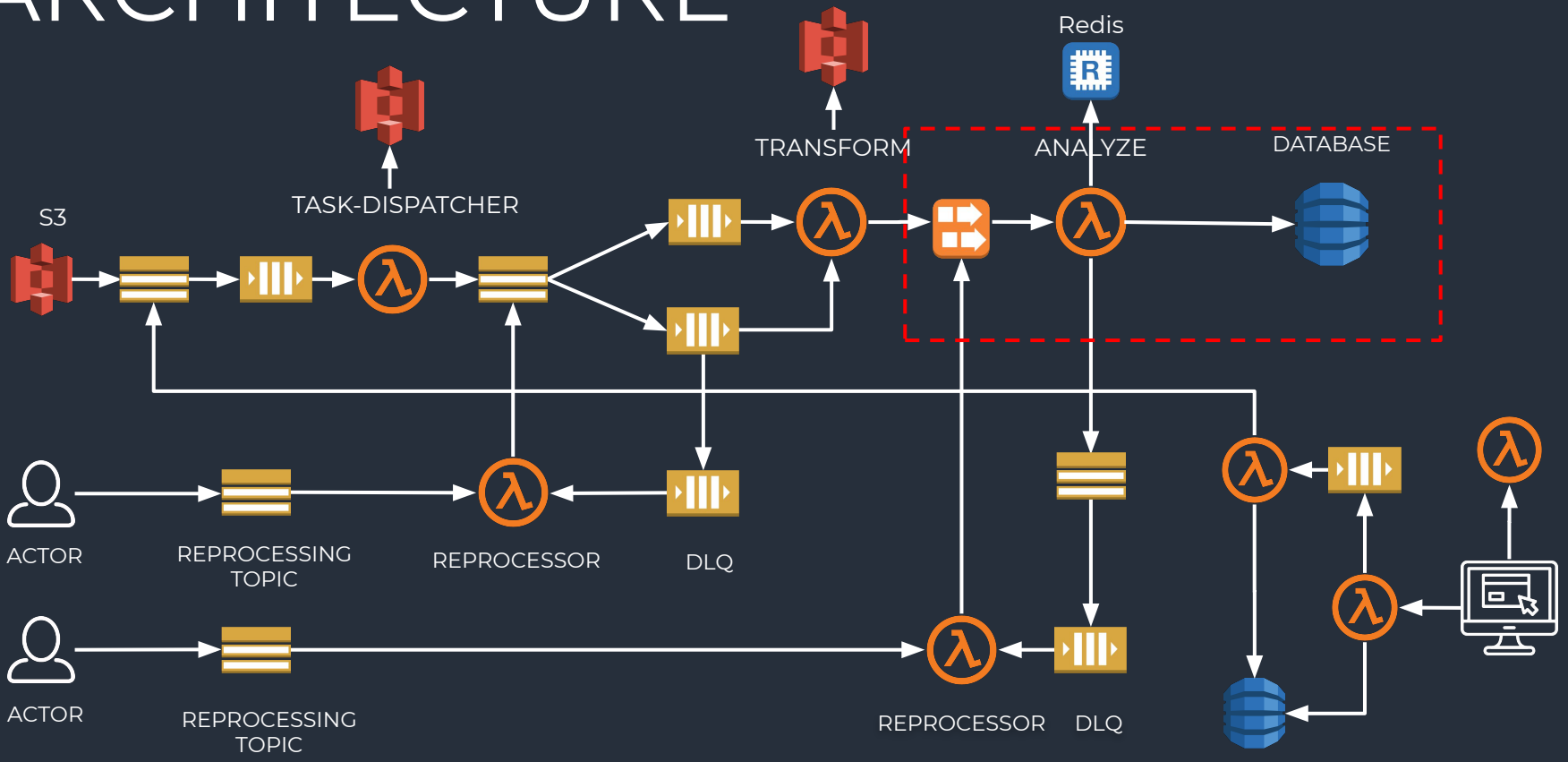


@matvi3nko

PROGRESS

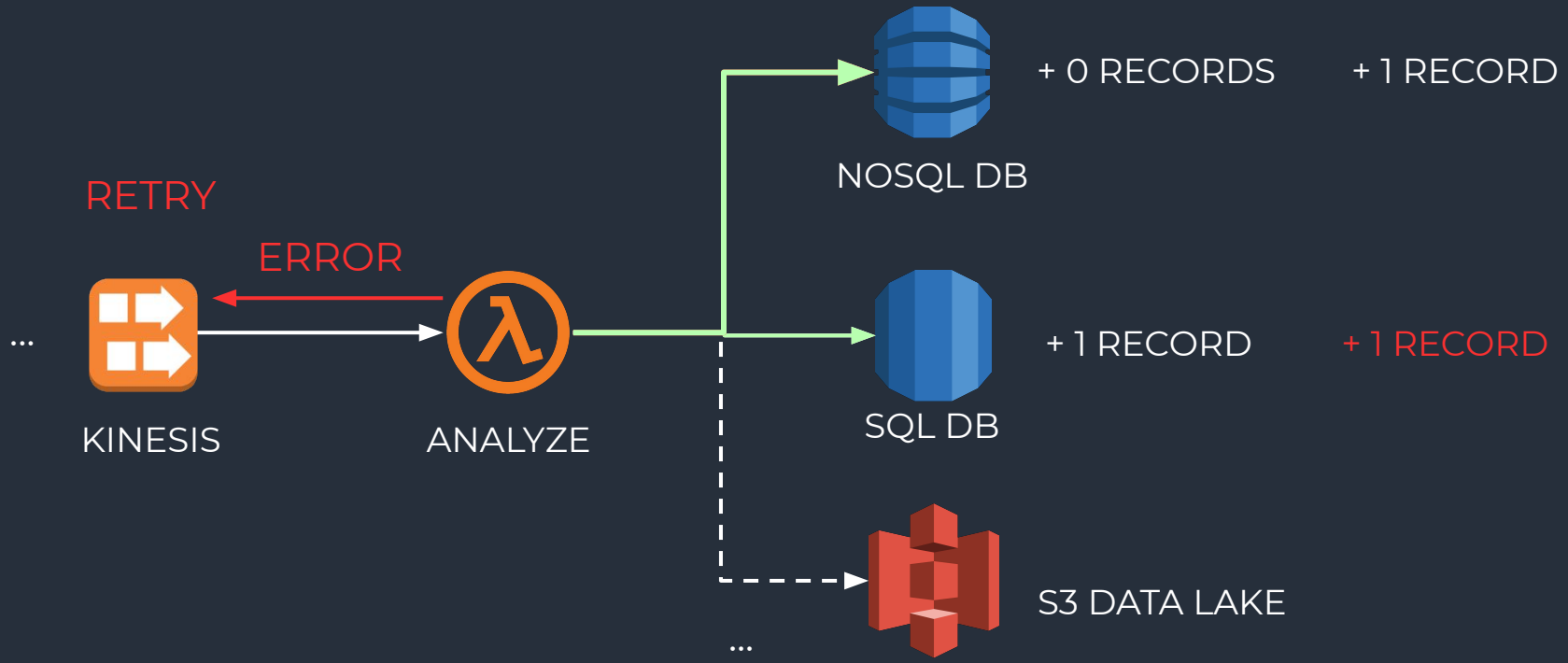

SCALABILITY
READ AND PROCESS DATA
ERROR HANDLING
SOLUTION STRUCTURE AND FUNCTION
BASE ARCHITECTURE DESIGN

ARCHITECTURE

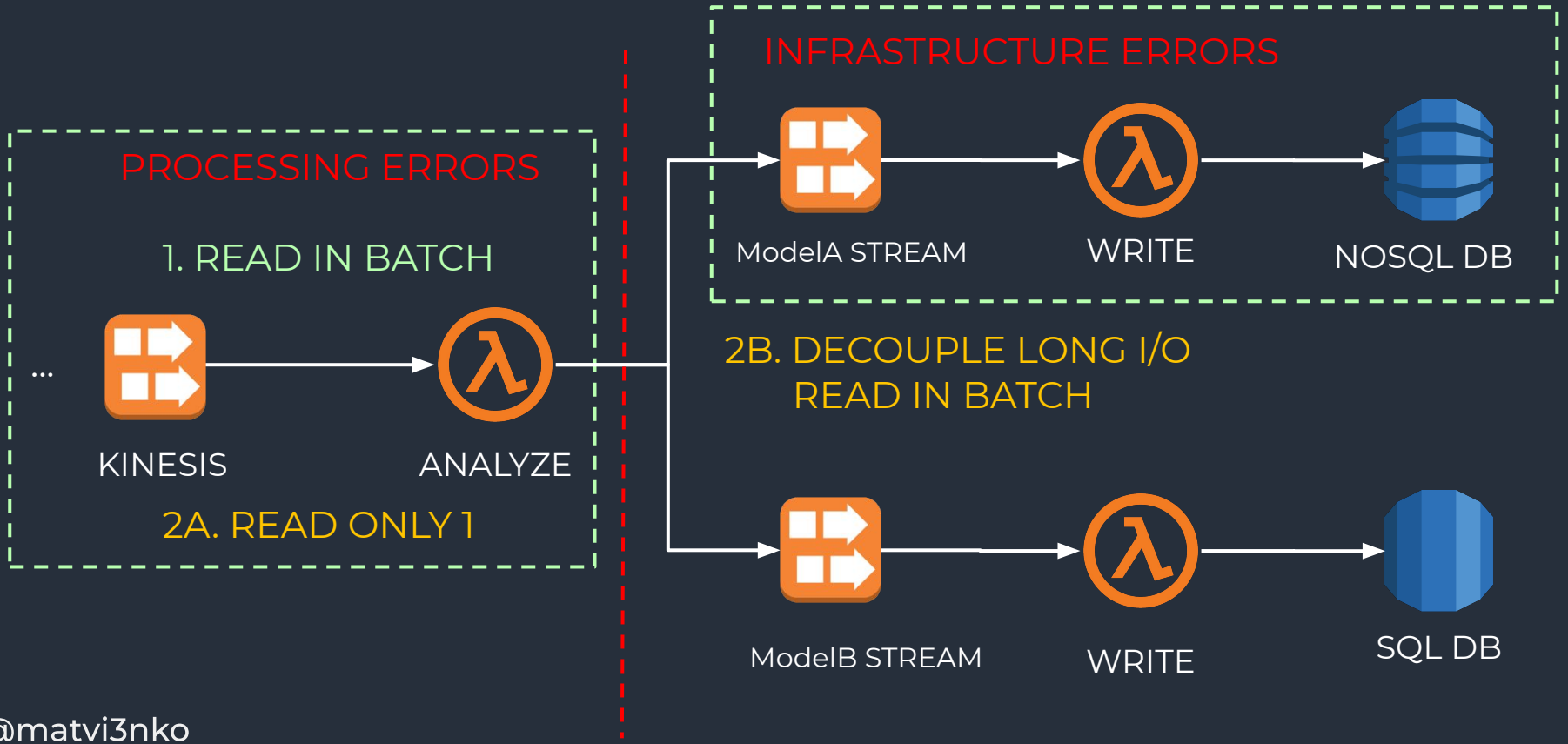


@matvi3nko

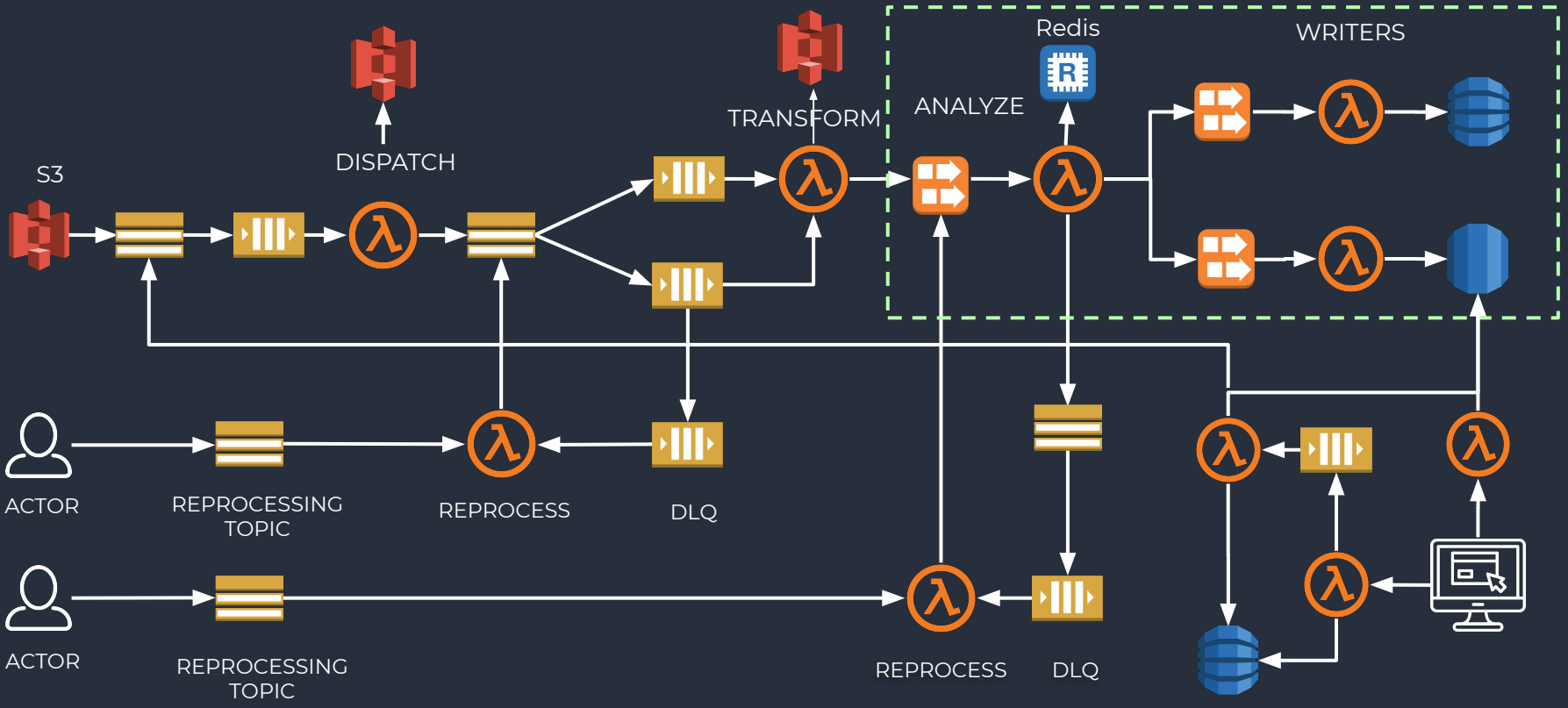
WRITING TO A DATABASES



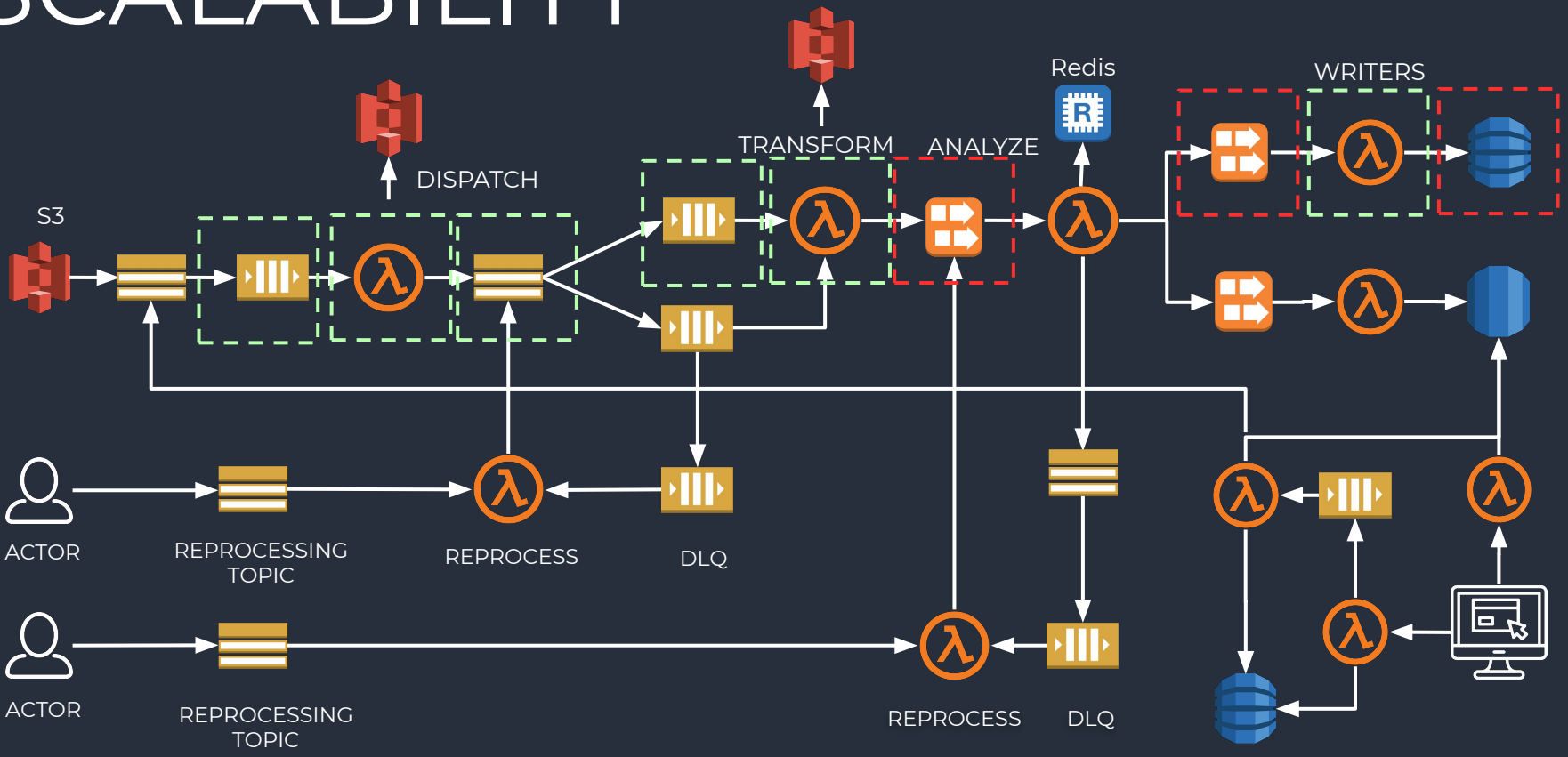
DECOUPLING


INFRASTRUCTURE ERRORS
PROCESSING ERRORS
1. READ IN BATCH
...
KINESIS
ANALYZE
2A. READ ONLY 1
2B. DECOUPLE LONG I/O
READ IN BATCH
ModelA STREAM
WRITE
NOSQL DB
ModelB STREAM
WRITE
SQL DB

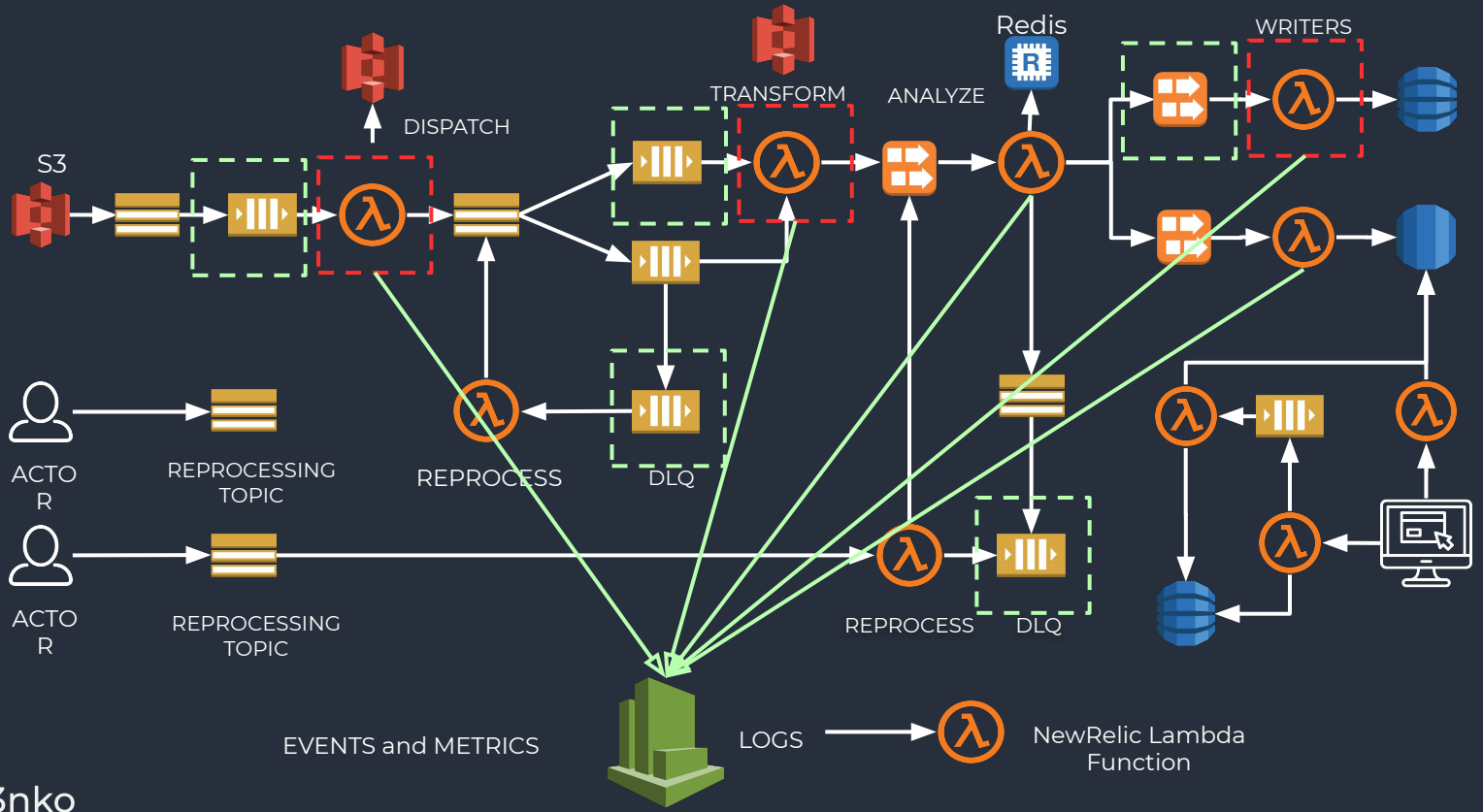
DECOUPLED WRITERS



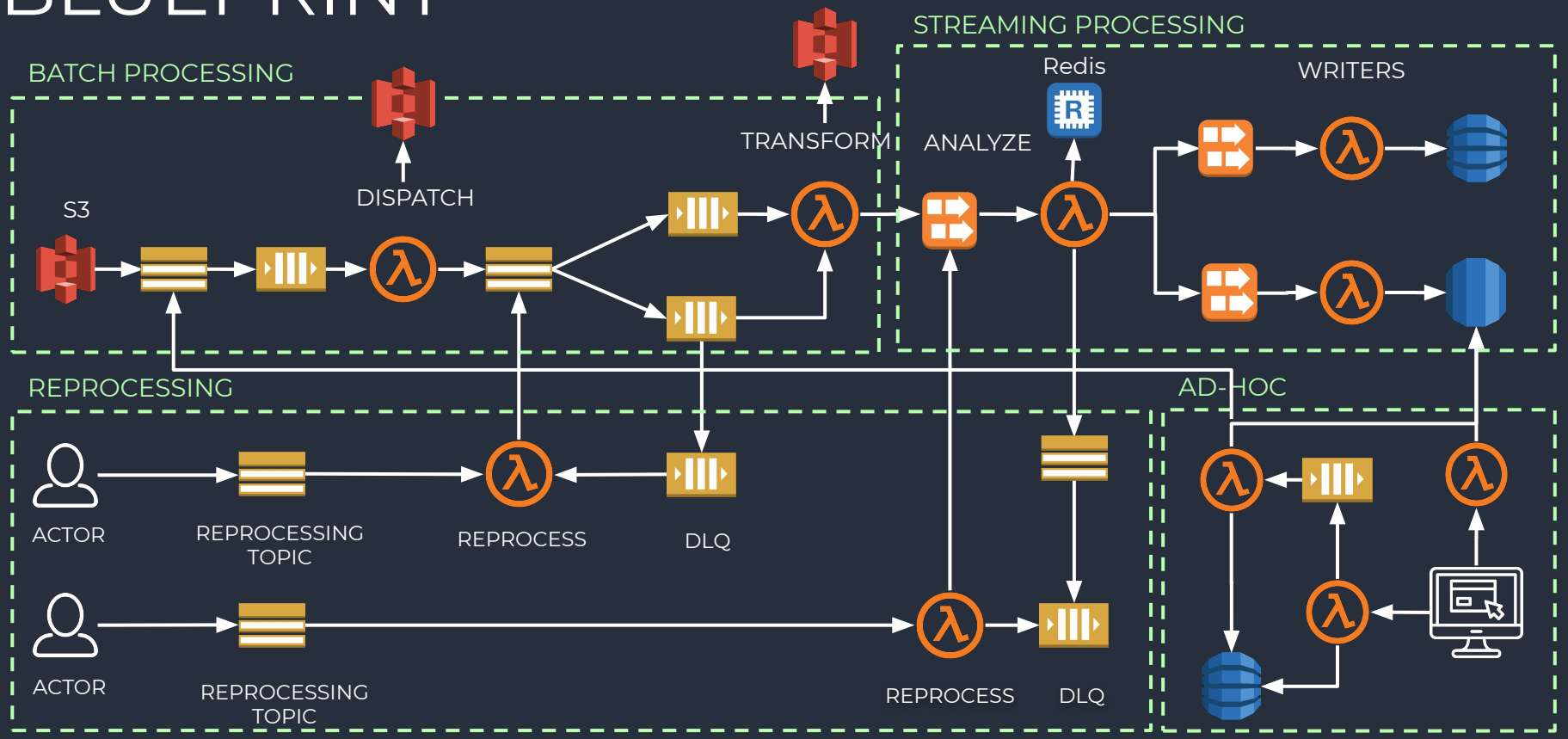
SCALABILITY



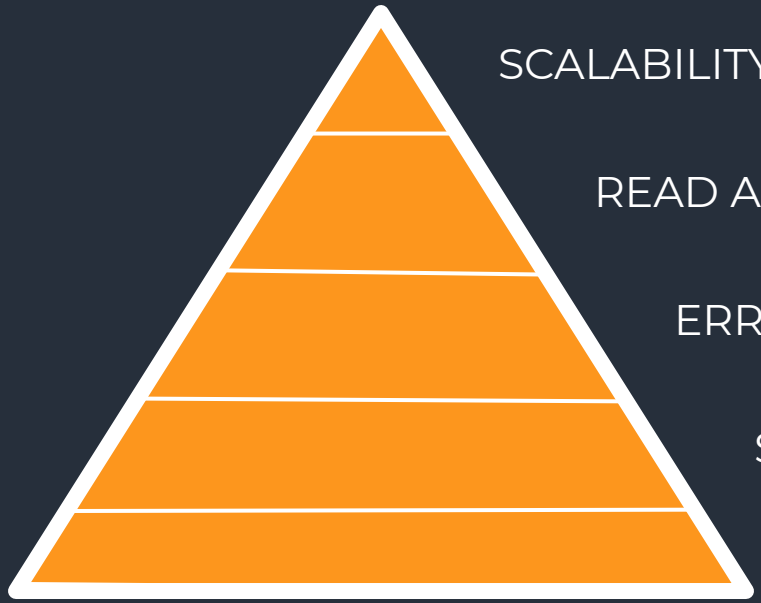
TROUBLESHOOTING



BLUEPRINT


BATCH PROCESSING
STREAMING PROCESSING
S3
DISPATCH
TRANSFORM
Redis
ANALYZE
WRITERS
REPROCESSING
AD-HOC
ACTOR
REPROCESSING TOPIC
REPROCESS
DLQ
ACTOR
REPROCESSING TOPIC
REPROCESS
DLQ

PROGRESS


SCALABILITY
READ AND PROCESS DATA
ERROR HANDLING
SOLUTION STRUCTURE AND FUNCTION
BASE ARCHITECTURE DESIGN

SUMMARY

1. Highly flexible architecture for changes in a result.
2. Massive scalable up to 5K, 7K, **10K lambda functions in parallel**
3. Terabytes of data
4. Serverless vs EC2: 60% – 5 times cheaper
5. Developer role in architecture

THANKS!

Nikolay Matvienko

matvi3nko@gmail.com

Twitter.com/**matvi3nko**

github.com/**matvi3nko**