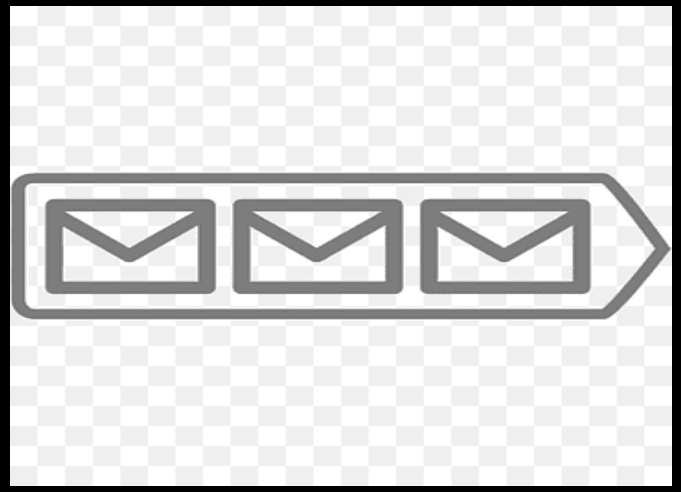
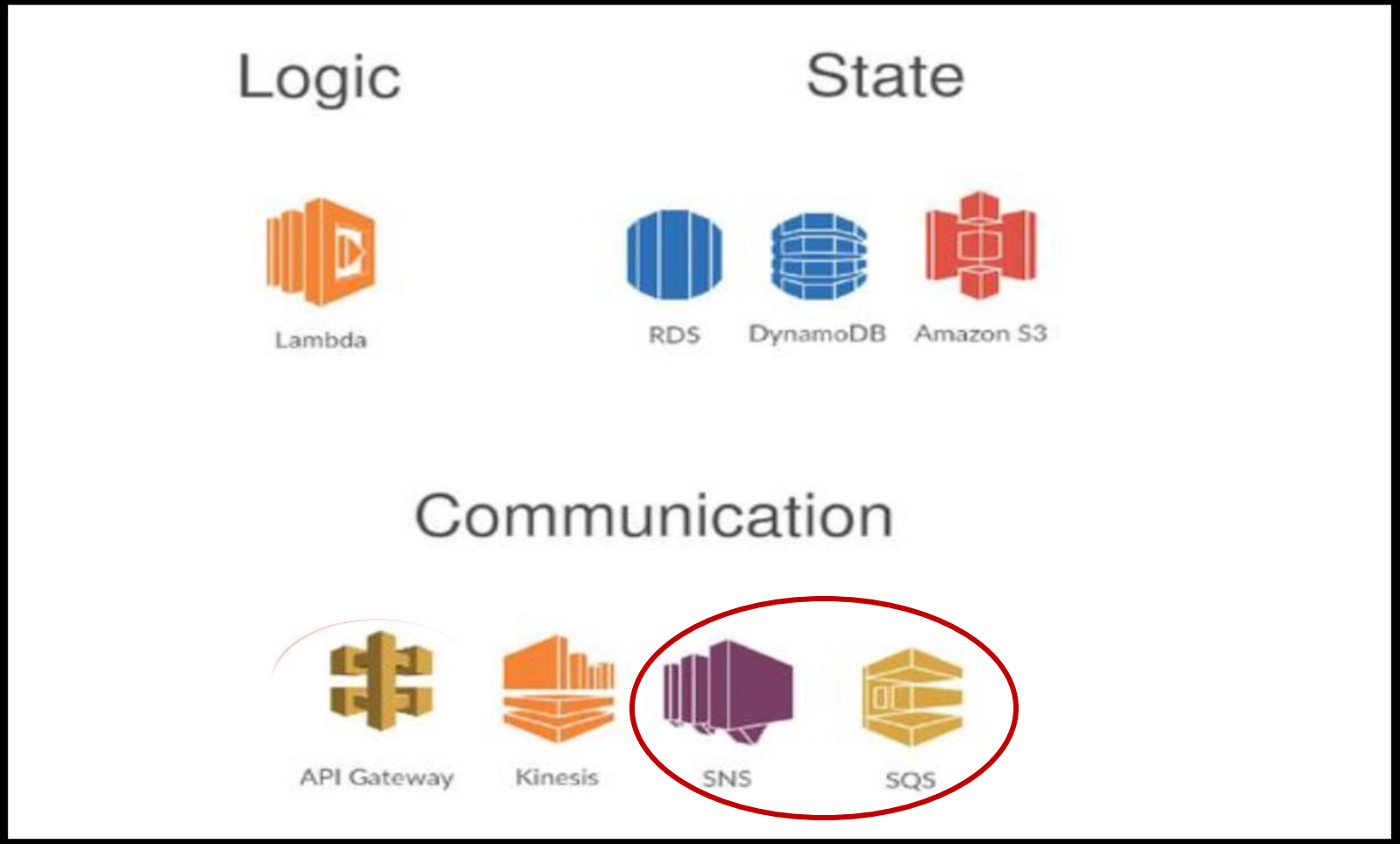
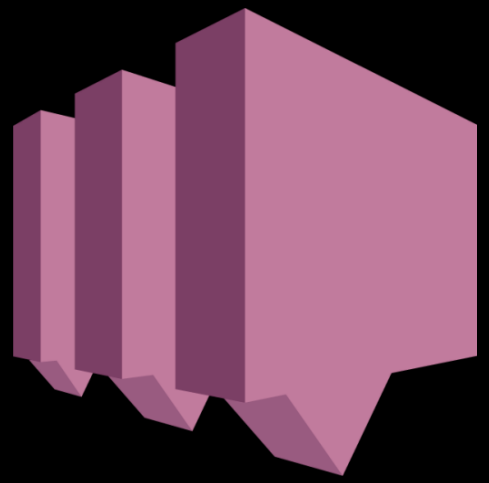


AWS Integration and Messaging Services (Contd).

Components of a Serverless, Message-Driven application

(aka Event Driven Architecture - EDA)





Simple Notification Service (SNS)

Amazon SNS

- Released in 2010.
- A ‘serverless’ publish-subscribe (pub/sub) messaging service.
- When you want to send a message to many receivers.
  - SQS is point-to-point.
  - SNS is pub/sub.
- The publisher sends a message to an SNS topic.
- Many subscribers can listen to a topic.
- Each topic subscriber gets all the messages.
- Subscribers can be:
  - SQS, HTTP / HTTPS, Lambda function
  - Emails (SES)
  - SMS messages, Mobile Notifications

SNS - Features

- Integrates with lots of services (Publishers):
  - Lambda.
  - S3 (Bucket change notifications).
  - Cloudwatch (Alarm notification).
  - etc
- Encryption:
  - In-flight encryption using HTTPS API.
  - At-rest encryption using KMS keys.
- Message Filtering:
  - Subscriber can declare a filtering policy to limit the messages it receives to those of interest.

SNS - Features

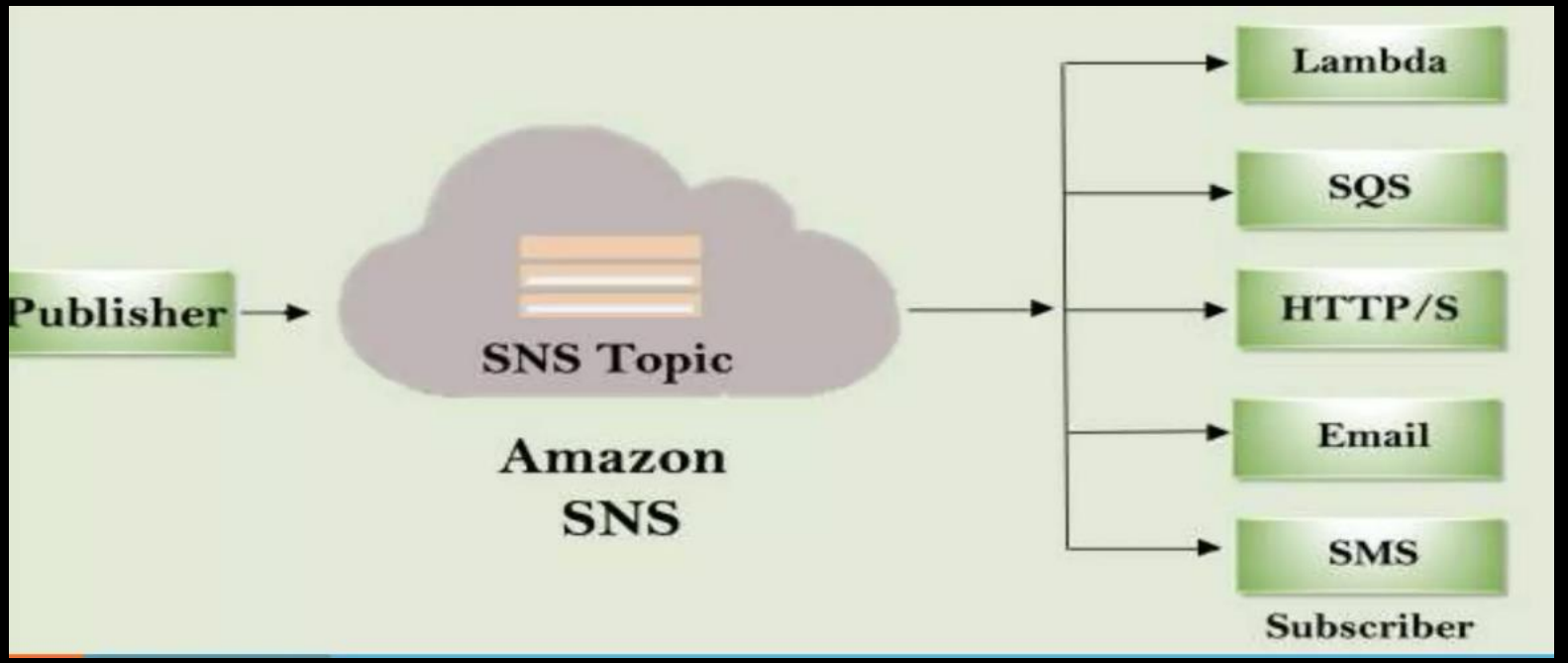
- Security:
  - Access Controls: IAM policies to regulate access to the SNS API.
  - SNS Access Policies (similar to S3 bucket policies):
    - Cross-account access to SNS topics.
    - Allowing other services (e.g. S3) to write to an SNS topic.

- Auto-scaling.

- DLQ – an SQS queue for messages that can't be delivered to a subscriber due to client errors or server errors.

Topics

- An SNS topic is a logical access point that acts as a communication channel.
- A topic lets you group multiple endpoints, e.g. SQS, Lambda, SMS



Demo – CDK provisioning code

- Architecture:

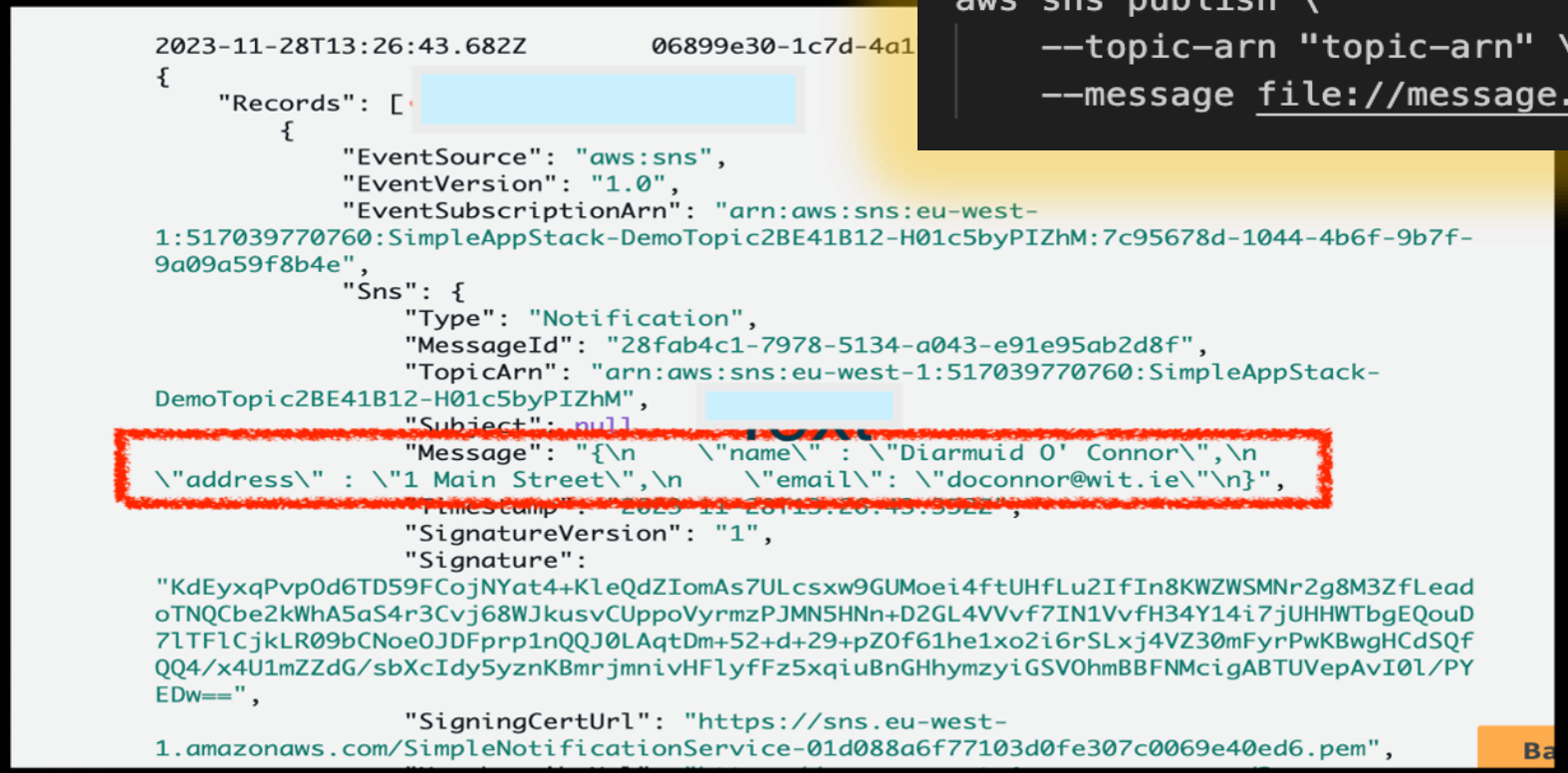
  AWS CLI (Publisher) -→ SNS Topic -→ Lambda (Subscriber)

```
23        const demoTopic = new sns.Topic(this, "DemoTopic", {
24          displayName: "Demo topic",
25        });
26
27        const processMessageFn = new lambdanode.NodejsFunction(
28          this,
29  ›       "processMsgFn", …
35          }
36        );
37
38        demoTopic.addSubscription(new subs.LambdaSubscription(processMessageFn));
39
40        new cdk.CfnOutput(this, "topicARN", {
41          value: demoTopic.topicArn,
42        });
43
```

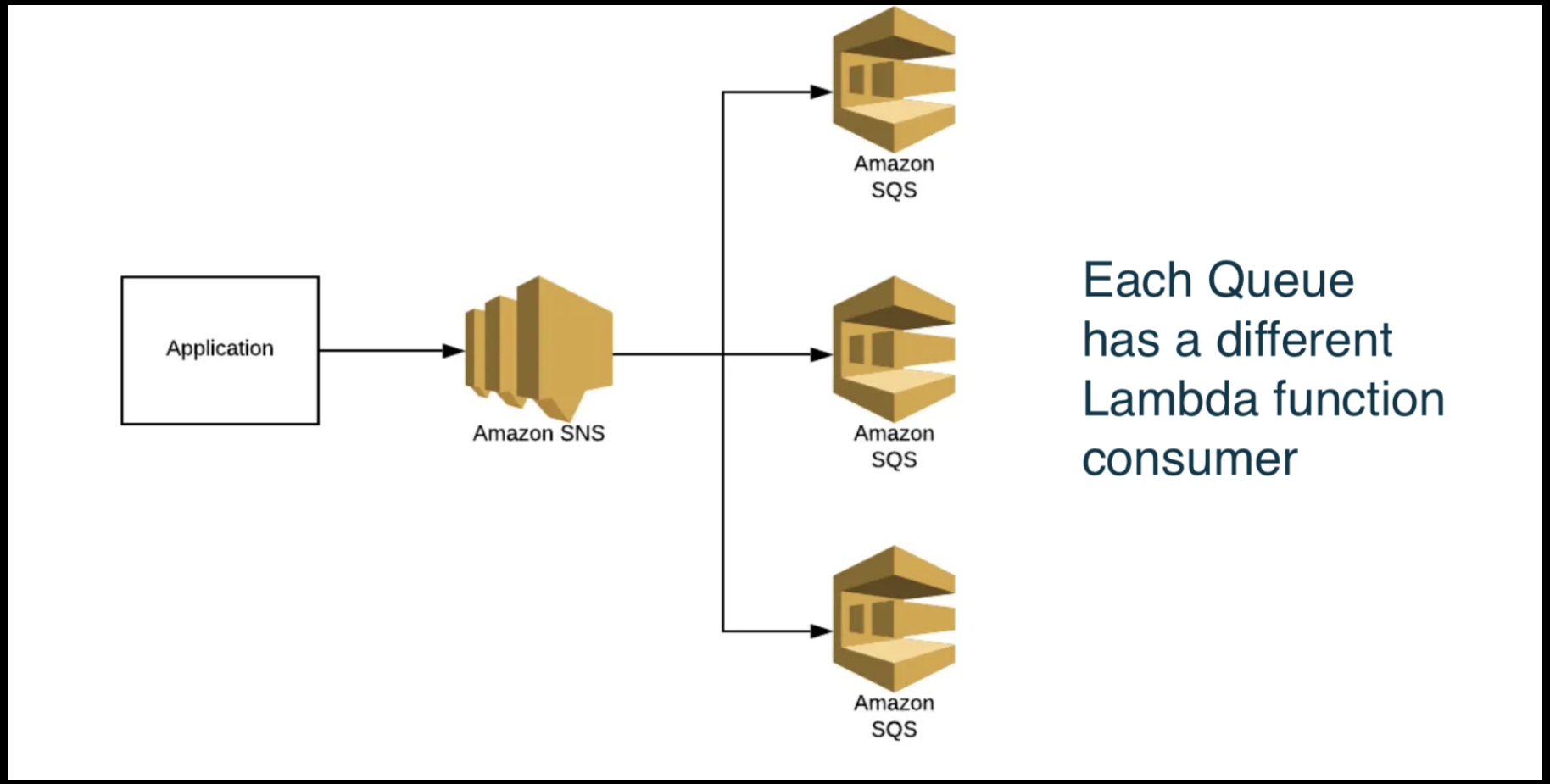
Demo – Lambda subscriber

- A Lambda subscriber receives a batch (?) of messages in its event parameter.

```
aws sns publish \
    --topic-arn "topic-arn" \
    --message file://message.json
```



The Fan-out pattern



Demo – Fan Out.

- The Fan Out subscribers can be a mixture of types.
- Demo Architecture:

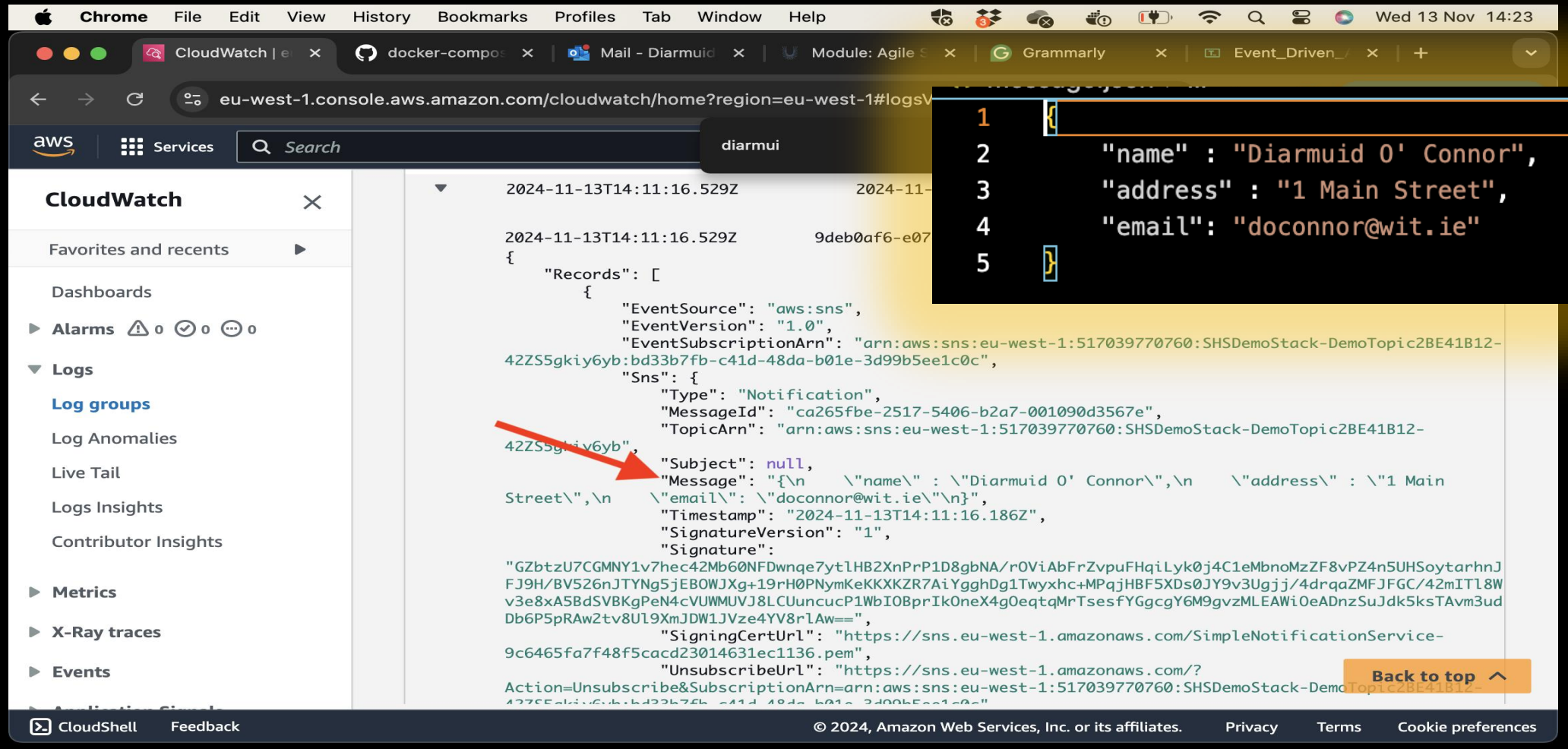
  AWS CLI (Pub) -→ SNS Topic -→ Lambda (Sub)

  -→ SQS (Sub). → Lambda (Consumer)

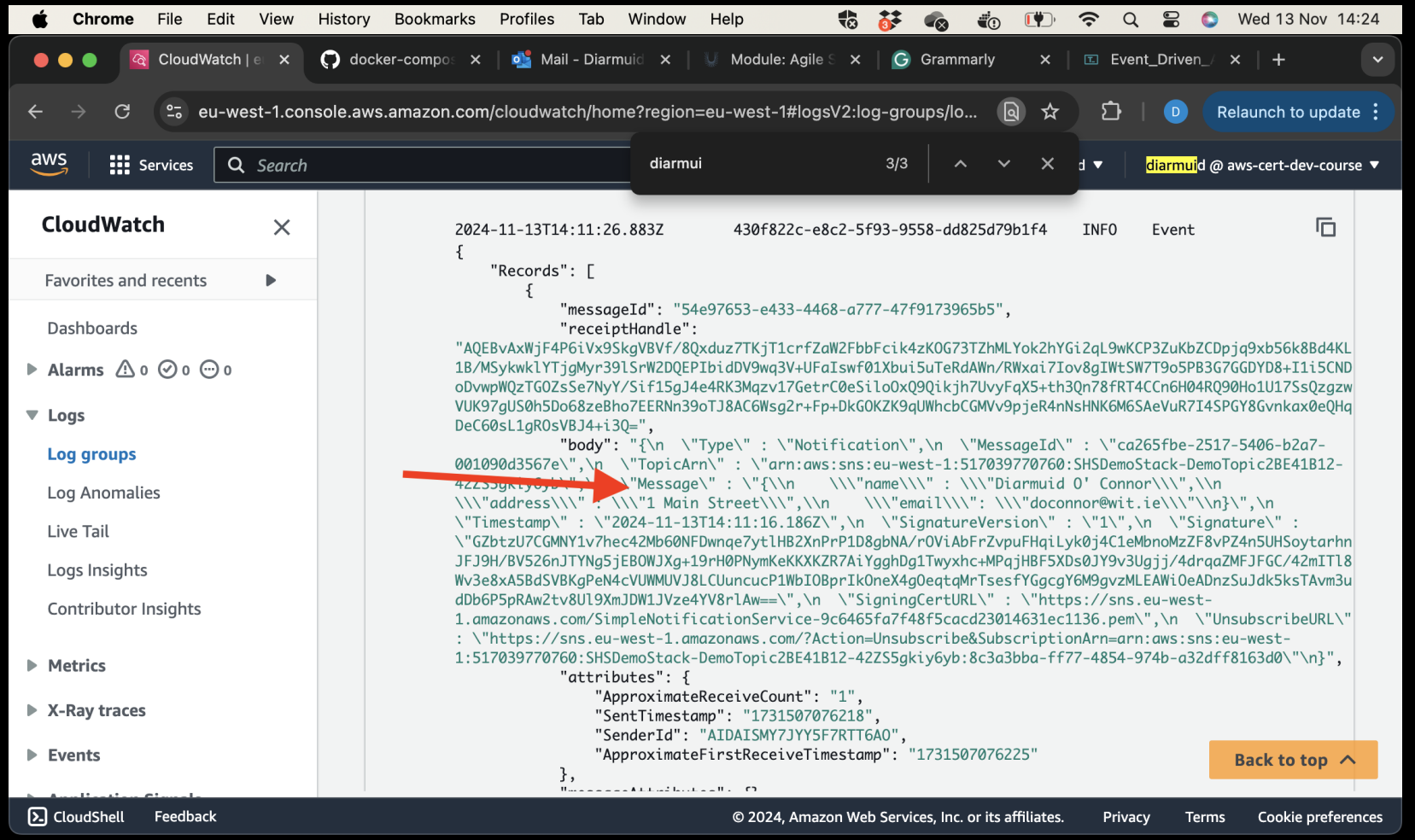
```
const demoTopic = new sns.Topic(this, "DemoTopic", {});
const queue = new sqs.Queue(this, "all-msg-queue", {});
const processSNSMessageFn = new lambdanode.NodejsFunction(
  this,
  "processSNSMsgFn",
  {
   ... properties .....
  }
);
// Subscribers
demoTopic.addSubscription(new subs.LambdaSubscription(processSNSMessageFn,
  {   ... properties ..... }));

demoTopic.addSubscription(new subs.SqsSubscription(queue,
  { ... properties ..... }));
```

Demo – The Lambda subscriber event parameter



Demo – The Lambda Q consumer event parameter



The SNS envelope.

- SNS wraps the source message in an envelope before sending it to an SQS queue subscriber.
  - Configurable

```
demoTopic.addSubscription(new subs.SqsSubscription(queue, {
  rawMessageDelivery: true,
}));
```

The SNS envelope.

The SNS envelope.

Lambda Vs { SQS → Lambda } subscribers

- Disadvantages (Lambda subscriber)
  - No batching is available when processing messages from SNS.
  - No control of Lambda Concurrency,
    - i.e. messages are processed one by one as soon as they arrive.
  - Lambda function is responsible for handling errors/retries
    - A DLQ needs to be configured.

- Advantages (Lambda subscriber):
  - Good for time-critical processing.

SNS - Delivery protocols and policies.

- SNS defines a delivery policy for each delivery protocol (subscriber type).
- The policy defines how SNS retries the delivery of messages when server-side errors occur,
    - i.e. when the service that hosts the subscriber is unavailable/not responding.

. When the delivery policy is exhausted, SNS stops retrying and discards the message.

    - A DLQ can be assigned for this case.

SNS - Delivery protocols and policies.

Endpoint type	Delivery protocols	Immediate retry (no delay) phase	Pre-backoff phase	Backoff phase	Post-backoff phase	Total attempts
AWS managed endpoints	Amazon Kinesis Data Firehose[1]	3 times, without delay	2 times, 1 second apart	10 times, with exponential backoff, from 1 second to 20 seconds	100,000 times, 20 seconds apart	100,015 times, over 23 days
	AWS Lambda					
	Amazon SQS					
Customer managed endpoints	SMTP	0 times, without delay	2 times, 10 seconds apart	10 times, with exponential backoff, from 10 seconds to 600 seconds (10 minutes)	38 times, 600 seconds (10 minutes) apart	50 attempts, over 6 hours
	SMS					
	Mobile push					

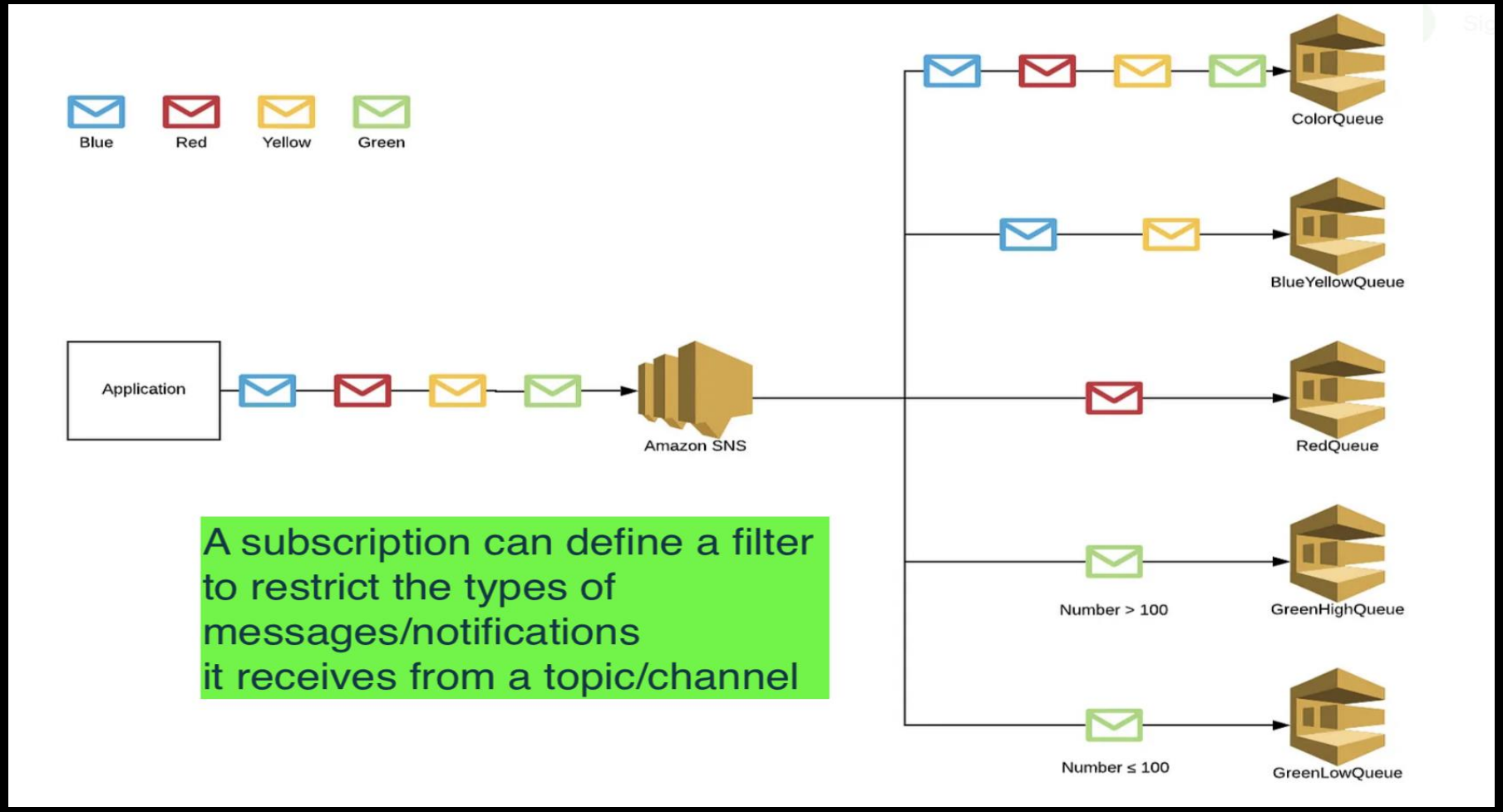
Lambda subscriber DLQ

- SNS invokes a lambda function subscriber asynchronously.
  - SNS does not wait for a response.
    $\Rightarrow$ Lambda service must handle function failures cases.

- Ex.: Architecture:

AWS CLI (Pub) → SNS Topic → Lambda (Subscriber)
|
|→ DLQ → Lambda (Consumer)

Fan Out pattern - Filtering.



Filtering

- The Filtering policy can be based on:
    1. Attributes of the message or
    2. The message body.

- Filtering criteria options:
    1. String Filter.
        - Conditions - allowList, denyList, matchPrefixes
    2. Numeric Filter.
        - Conditions – allowList, greaterThan, lessThan, between.
    3. Exists Filter.

→ DLQ → Lambda (Consumer)

Demo – Message Attribute Filtering

- Architecture:

  AWS CLI (Pub) → SNS Topic → <Filter> → Lambda (Sub)

  → <Filter> → SQS Q -→ Lambda (Con)

- Filter policy - The Lambda subscriber should only receive messages with a user_type attribute set to Student or Lecturer.

Demo – Message Attribute Filtering

```
demoTopic.addSubscription(
  new subs.LambdaSubscription(processSNSMessageFn, {
    filterPolicy: {
      user_type: sns.SubscriptionFilter.stringFilter({
        allowlist: ["Student", "Lecturer"],
      }),
    },
  })
);
```

{} attributes.json > ...

```
{
    "user_type": {
        "DataType": "String",
        "StringValue": "Lecturer"
    },
    "source": {
        "DataType": "String",
        "StringValue": "Moodle2023"
    }
}
```

```
aws sns publish --topic-arn topic-arn-value \
    --message-attributes file://attributes.json \
    --message file://message.json
```

{} message.json > ...

```
{
    "name" : "Diarmuid O' Connor",
    "address" : "1 Main Street",
    "email": "doconnor@wit.ie"
}
```

Demo – Message Attribute Filtering

To Be Continued