AWS

SUMMIT

Deep Dive on AWS IoT

Olawale Oladehin, Sr. Solutions Architect, IoT

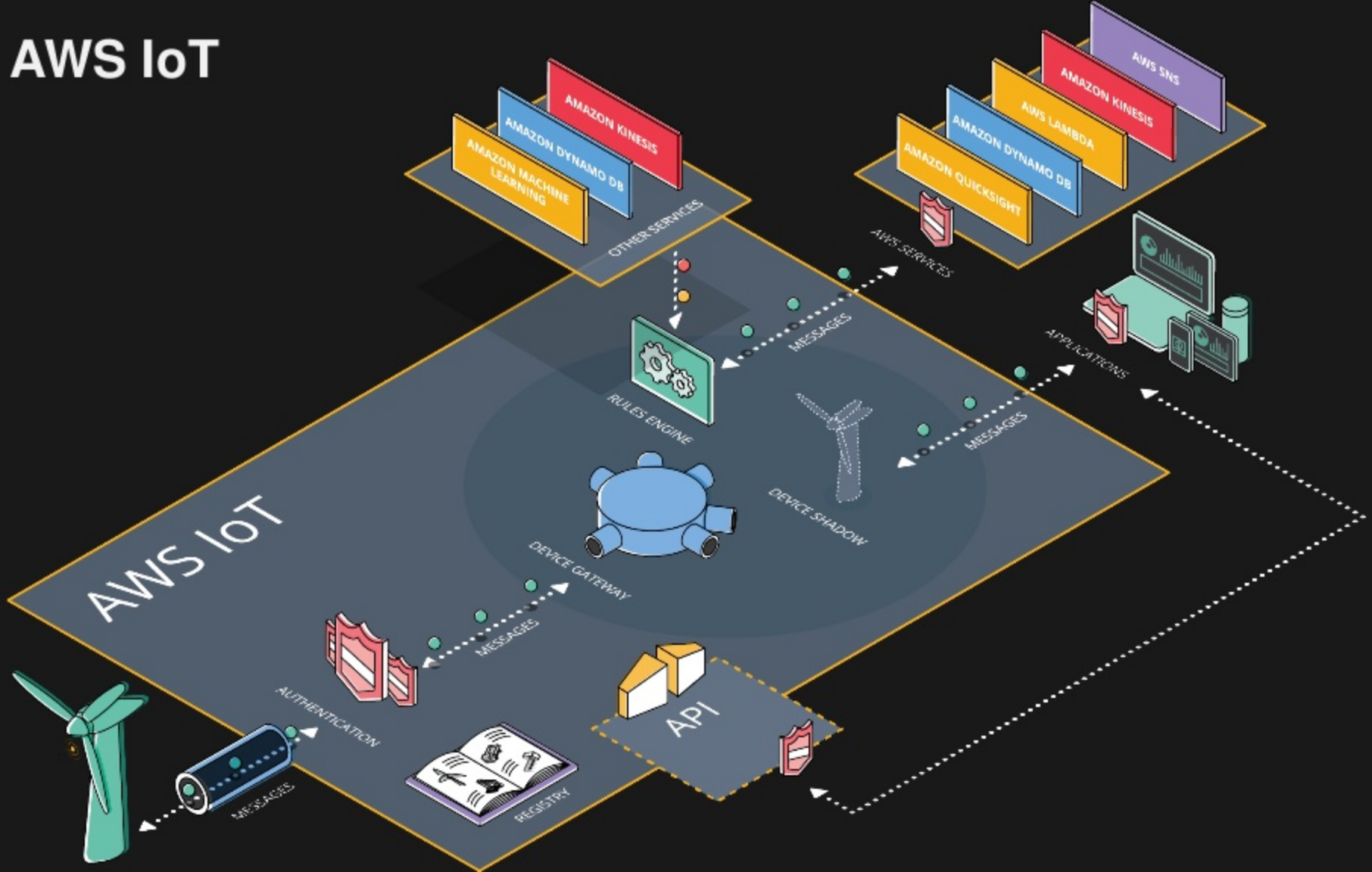
August 14, 2017

amazon web services

Agenda

- Overview of AWS IoT
- Telemetry & analytics
- Cloud control
- Mobile control
- Lifecycle management
- Wrap-up

AWS IoT

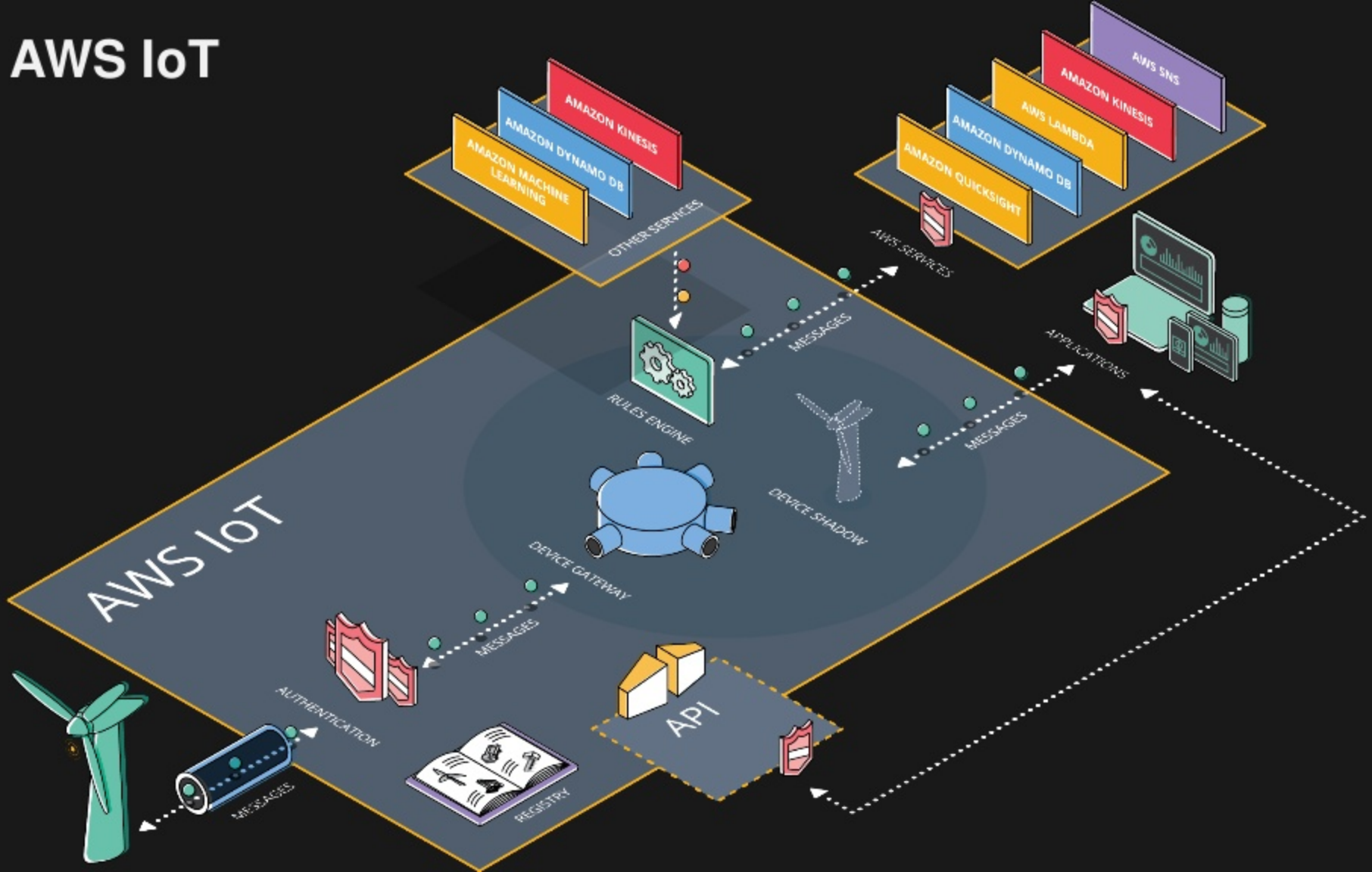

AMAZON KINESIS
AMAZON DYNAMO DB
AMAZON MACHINE LEARNING
OTHER SERVICES
AWS SNS
AMAZON KINESIS
AWS LAMBDA
AMAZON DYNAMO DB
AMAZON QUICKSIGHT
AWS SERVICES
MESSAGES
RULES ENGINE
APPLICATIONS
MESSAGES
DEVICE SHADOW
AWS IoT
DEVICE GATEWAY
MESSAGES
AUTHENTICATION
API
REGISTRY
MESSAGES

Telemetry & Analytics

AWS IoT Telemetry & Analytics

1. Connect devices
2. Send data
3. Collect and store the data
4. Do something with the data

AWS IoT


AMAZON KINESIS
AMAZON DYNAMO DB
AMAZON MACHINE LEARNING
OTHER SERVICES
AWS SNS
AMAZON KINESIS
AWS LAMBDA
AMAZON DYNAMO DB
AMAZON QUICKSIGHT
AWS SERVICES
MESSAGES
RULES ENGINE
APPLICATIONS
MESSAGES
DEVICE SHADOW
AWS IoT
DEVICE GATEWAY
MESSAGES
AUTHENTICATION
API
REGISTRY
MESSAGES

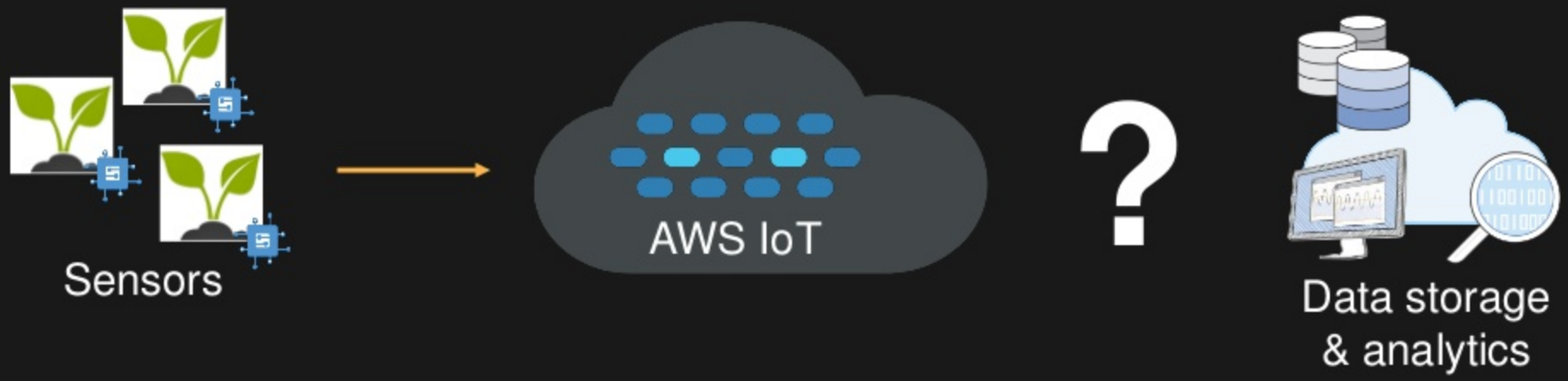
1) Connect the devices

1. Provision a certificate
2. Attach policy
3. Connect over MQTT

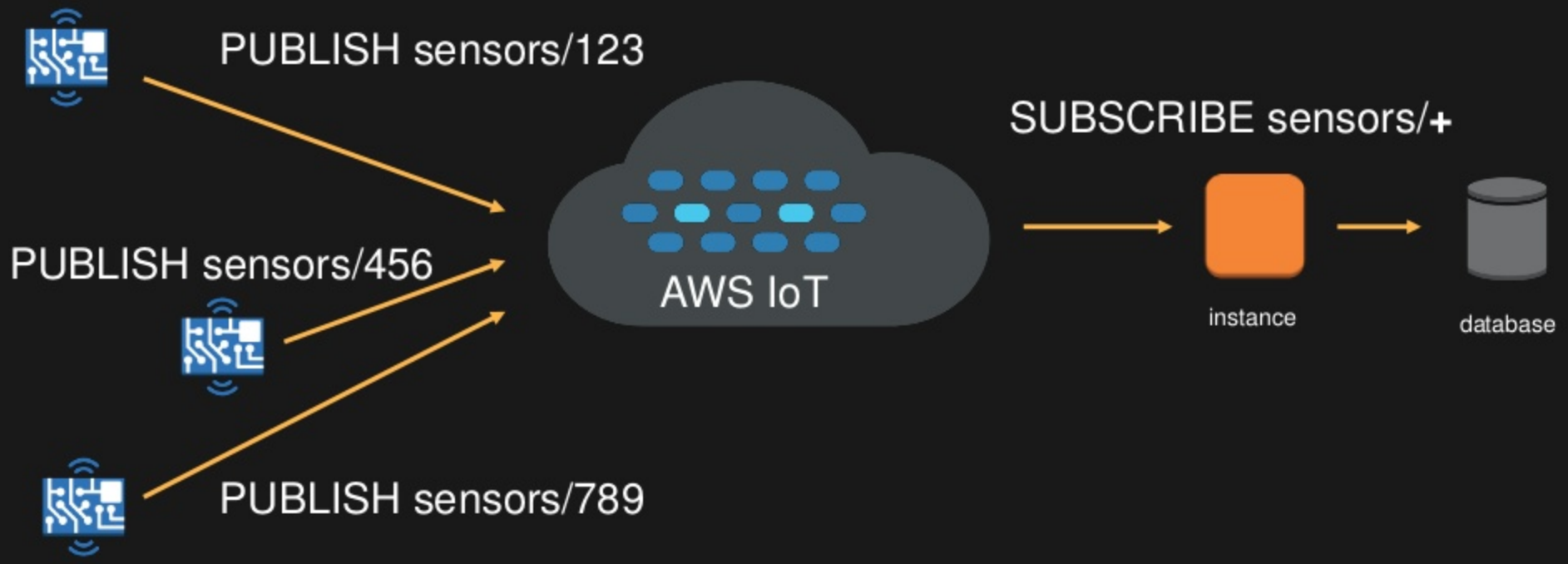
2) Send data

```
PUBLISH macdonald/sensors/123 (qos: 0)
  {
    "timestamp": "2016-01-29T10:00:00",
    "temperature": 55
    "humidity": 39,
    "ph": 6.7
  }
```

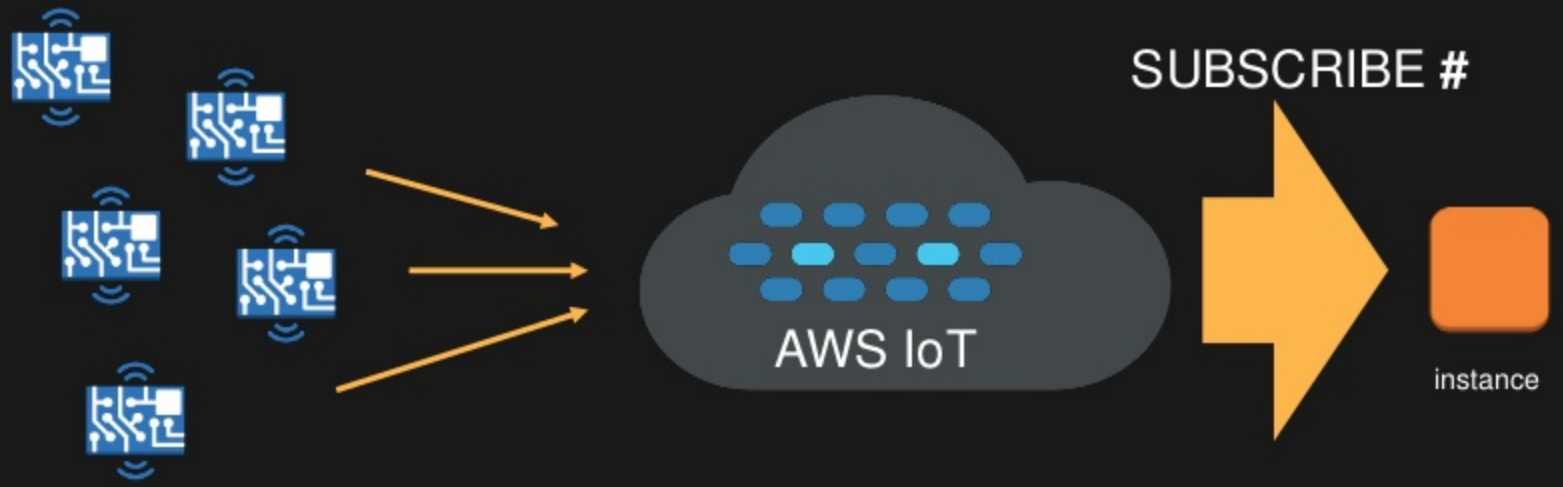
3) Collect the data



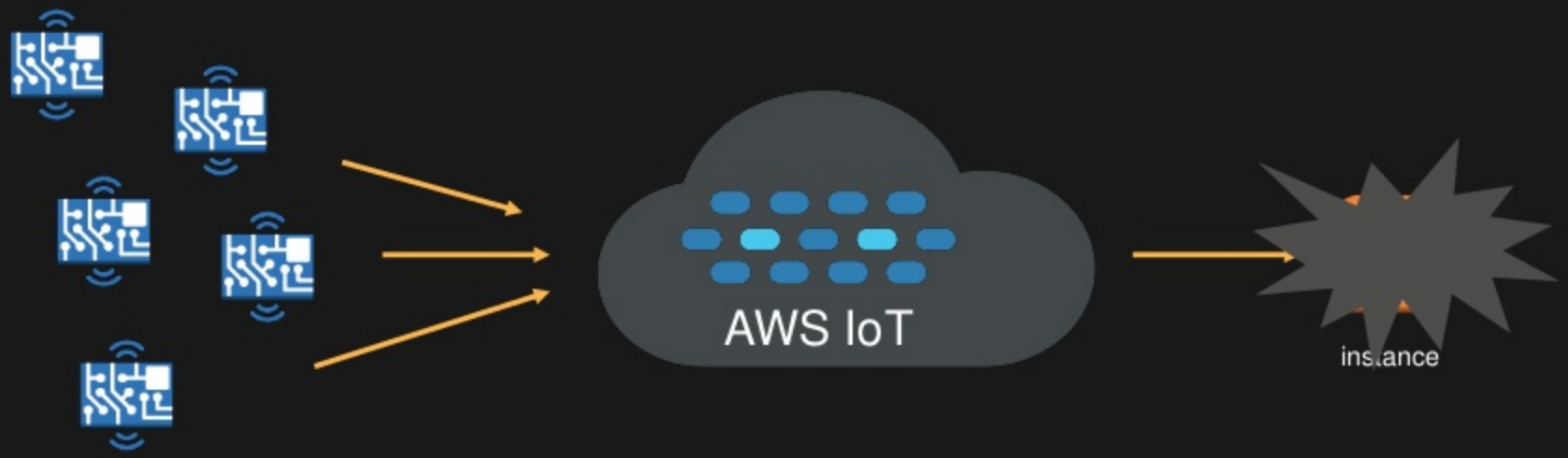
Single consumer (don't do this)



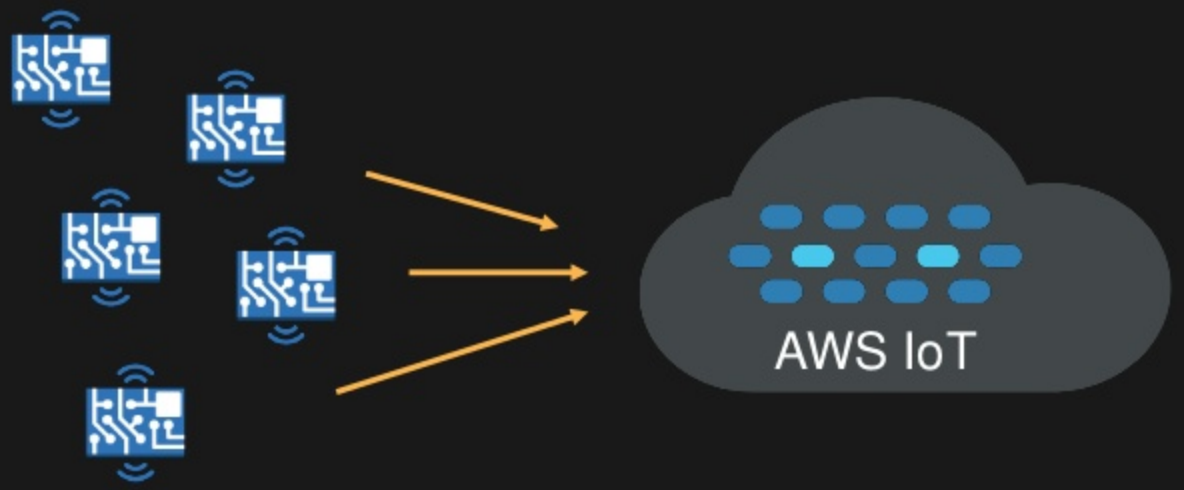
Don’t do this: scalability

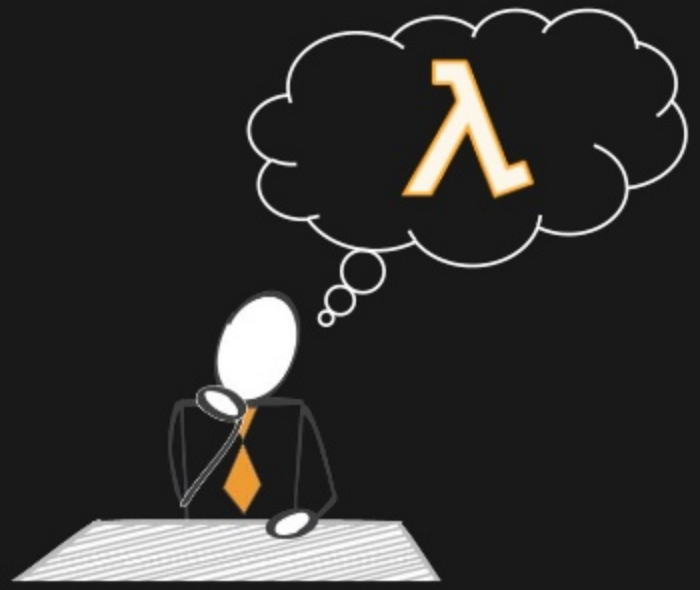


Don’t do this: availability

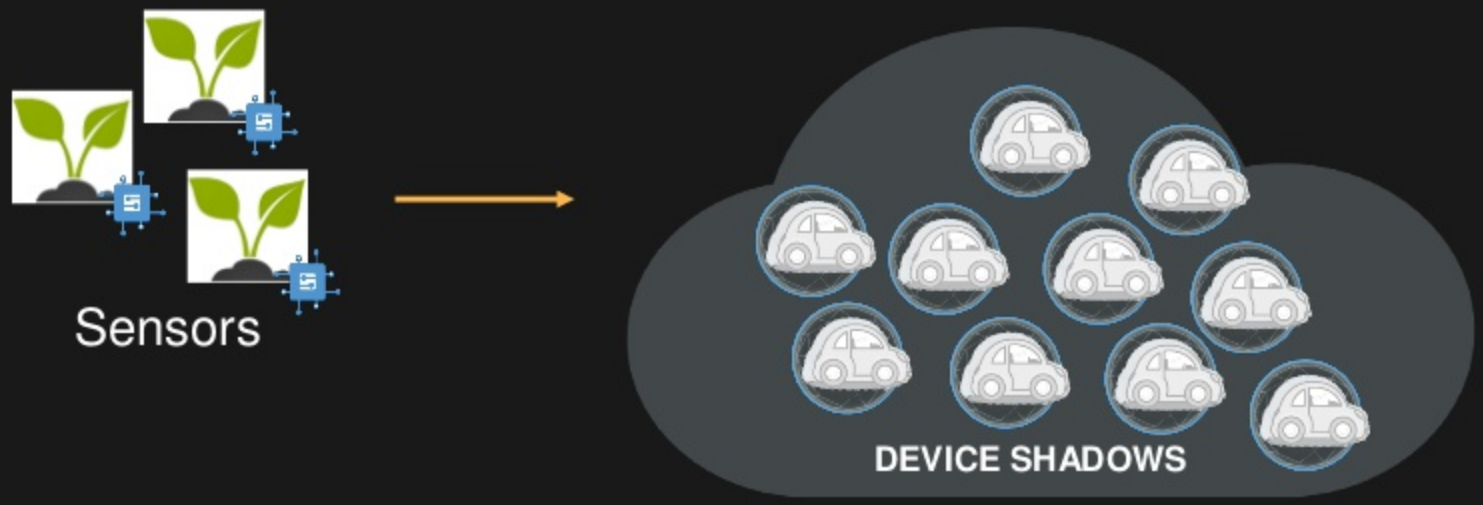


Don’t do this: maintainability





Store it in the device shadow (don’t do this)



AWS IoT Rules Engine

Rules Engine connects AWS IoT to external endpoints and AWS services.

Example rule

macdonald/sprinkler-456/temperature

```
{
   "rule": {
         "sql": "SELECT * AS message FROM 'macdonald/#'",
         "description": "Store all sensor data into dynamodb and firehose",
         "actions": [{
            "dynamoDB": {
                "tableName": "sensor_data",
                "roleArn": "arn:aws:iam::123456789012:role/aws_iot_dynamoDB",
                "hashKeyField": "sensor_id",
                "hashKeyValue": "${topic(2)}",
                "rangeKeyField": "timestamp“
                "rangeKeyValue": "${timestamp()}",
             }
         }, {
            "firehose": {
               "roleArn": "arn:aws:iam::123456789012:role/aws_iot_firehose",
               "deliveryStreamName": "my_firehose_stream"
            }
         }]
    }
}
```

Example rule

macdonald/sprinkler-456/temperature

```
{
   "rule": {
         "sql": "SELECT * AS message FROM 'macdonald/#'",
         "description": "Store all sensor data into dynamodb and firehose",
         "actions": [{
            "dynamoDB": {
                "tableName": "sensor_data",
                "roleArn": "arn:aws:iam::123456789012:role/aws_iot_dynamoDB",
                "hashKeyField": "sensor_id",
                "hashKeyValue": "${topic(2)}",
                "rangeKeyField": "timestamp“
                "rangeKeyValue": "${timestamp()}",
             }
         }, {
            "firehose": {
               "roleArn": "arn:aws:iam::123456789012:role/aws_iot_firehose",
               "deliveryStreamName": "my_firehose_stream"
            }
         }]
    }
}
```

Example rule

macdonald/sprinkler-456/temperature

```
{
   "rule": {
         "sql": "SELECT * AS message FROM 'macdonald/#'",
         "description": "Store all sensor data into dynamodb and firehose",
         "actions": [{
            "dynamoDB": {
                "tableName": "sensor_data",
                "roleArn": "arn:aws:iam::123456789012:role/aws_iot_dynamoDB",
                "hashKeyField": "sensor_id",
                "hashKeyValue": "${topic(2)}",
                "rangeKeyField": "timestamp“
                "rangeKeyValue": "${timestamp()}",
             }
         }, {
            "firehose": {
               "roleArn": "arn:aws:iam::123456789012:role/aws_iot_firehose",
               "deliveryStreamName": "my_firehose_stream"
            }
         }]
    }
}
```

Example rule

macdonald/sprinkler-456/temperature

```
{
   "rule": {
         "sql": "SELECT * AS message FROM 'macdonald/#'",
         "description": "Store all sensor data into dynamodb and firehose",
         "actions": [{
            "dynamoDB": {
                "tableName": "sensor_data",
                "roleArn": "arn:aws:iam::123456789012:role/aws_iot_dynamoDB",
                "hashKeyField": "sensor_id",
                "hashKeyValue": "${topic(2)}",
                "rangeKeyField": "timestamp“
                "rangeKeyValue": "${timestamp()}",
             }
         }, {
            "firehose": {
               "roleArn": "arn:aws:iam::123456789012:role/aws_iot_firehose",
               "deliveryStreamName": "my_firehose_stream"
            }
         }]
    }
}
```

IoT SQL functions

```
[…]
"hashKeyValue": "${topic(2)}",
“rangeKeyField": "timestamp“
"rangeKeyValue": "${timestamp()}”
[…]
```

- Math
  - abs, acos, asin, atan, atan2, cos, cosh, etc…
  - round, ceil, floor, exp,
- Bitwise Ops
  - bitand, bitor, bitxor, and bitnot
- String and Arrays
  - chr, concat, encode, endswith, get, indexof, length, lower, upper, etc…
  - newuuid, parse_time
- Hashing
  - md2, md5, sha1, sha224, sha256, sha384, sha512
- IoT
  - machinelearning_predict(modelId, roleARN, record)
  - get_thing_shadow(thingName, roleARN)
  - principal, topic, timestamp, traceid,clientid

Different data scenarios

Want to run a lot of queries constantly?

➡ Use Amazon Kinesis Firehose to write into Amazon Redshift

Need fast lookups, e.g., in Rules or Lambda functions?

➡ Write into DynamoDB, add indexes if necessary

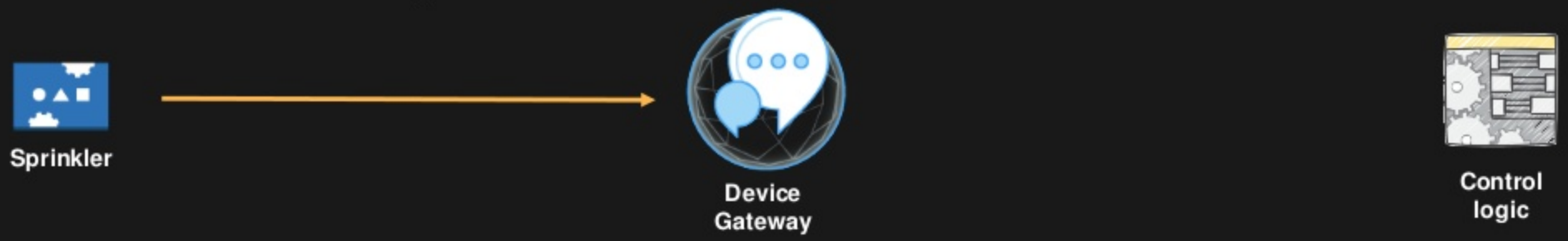
Have a need for heavy queries but not always-on?

➡ Use Firehose and S3, process with Amazon EMR or Athena.

Cloud Control

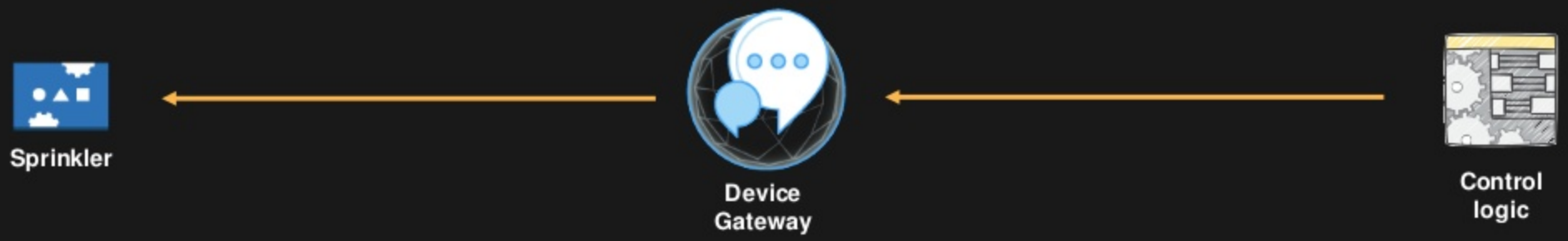
Publish on/off to the sprinkler

```
SUBSCRIBE
  macdonald/sprinkler-456
```



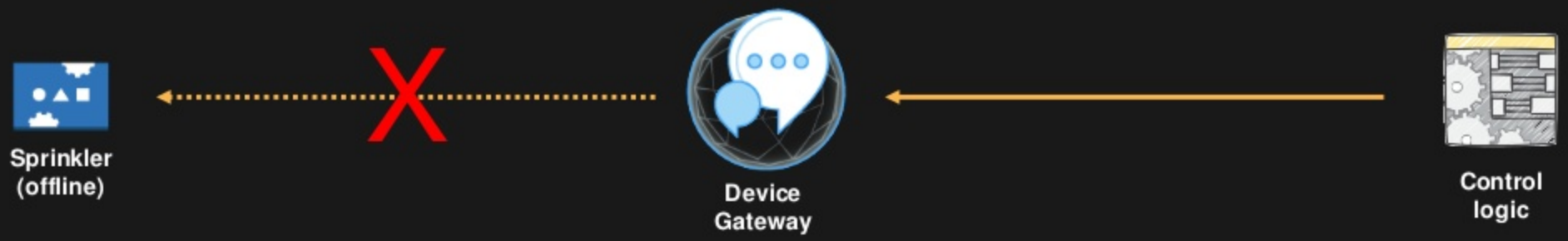
Publish on/off to the sprinkler

```
PUBLISH
  macdonald/sprinkler-456
  { "water": "on" }
```

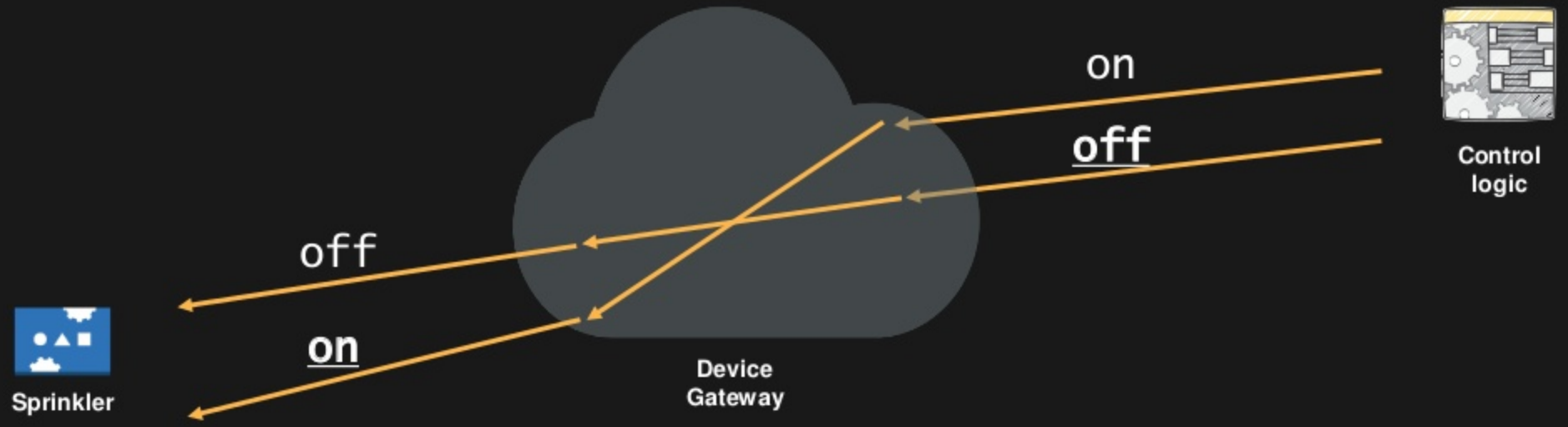


Direct publishing: why not?

```
PUBLISH
  macdonald/sprinkler-456
  { "water": "on" }
```



Direct publishing: why not?

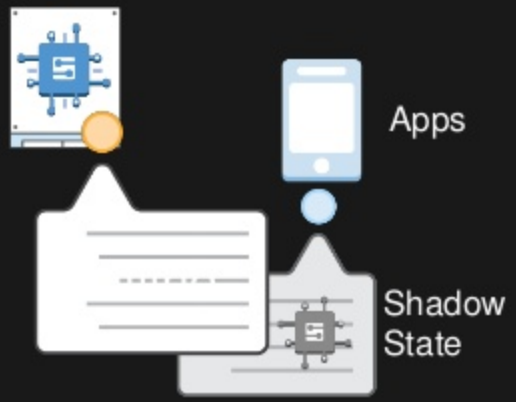


Direct publishing: why not?

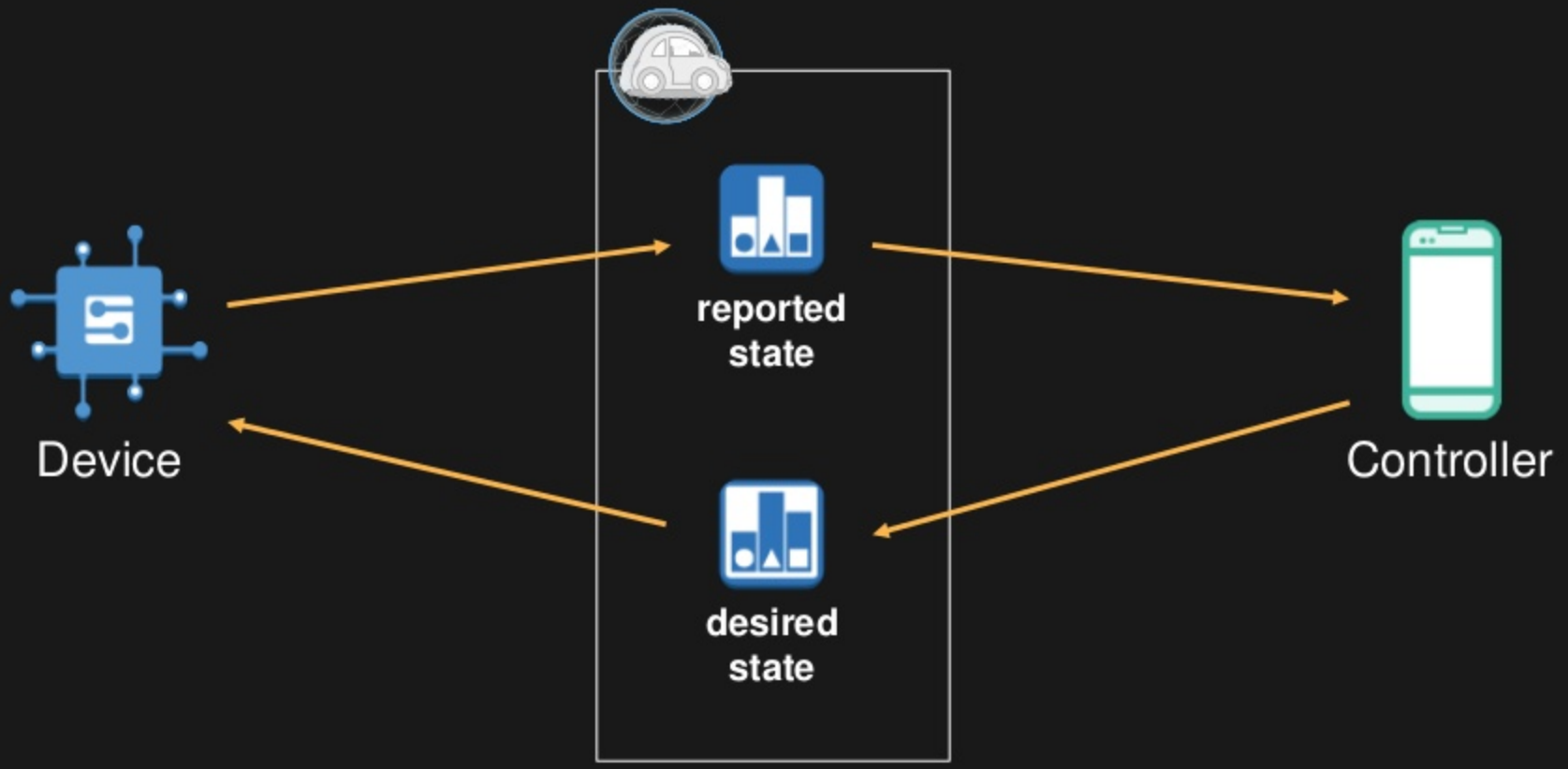
- Connection blips
- Messages aren’t ordered

So then what?

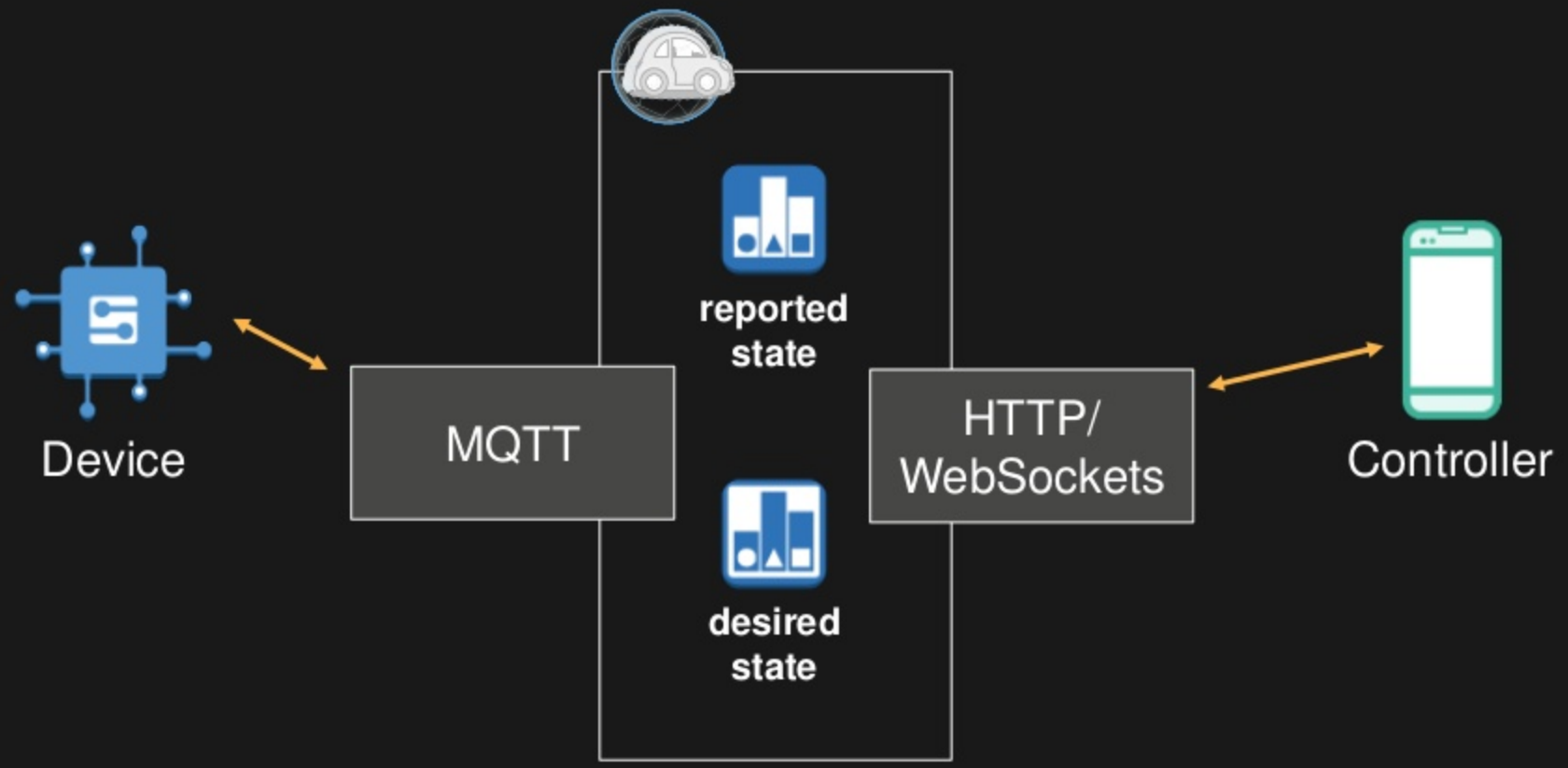
Device Shadows



Device Shadows



Device Shadows



AWS IoT Shadow - simple yet powerful

Thing

Report its current state to one or multiple shadows
Retrieve its desired state from shadow

Shadow

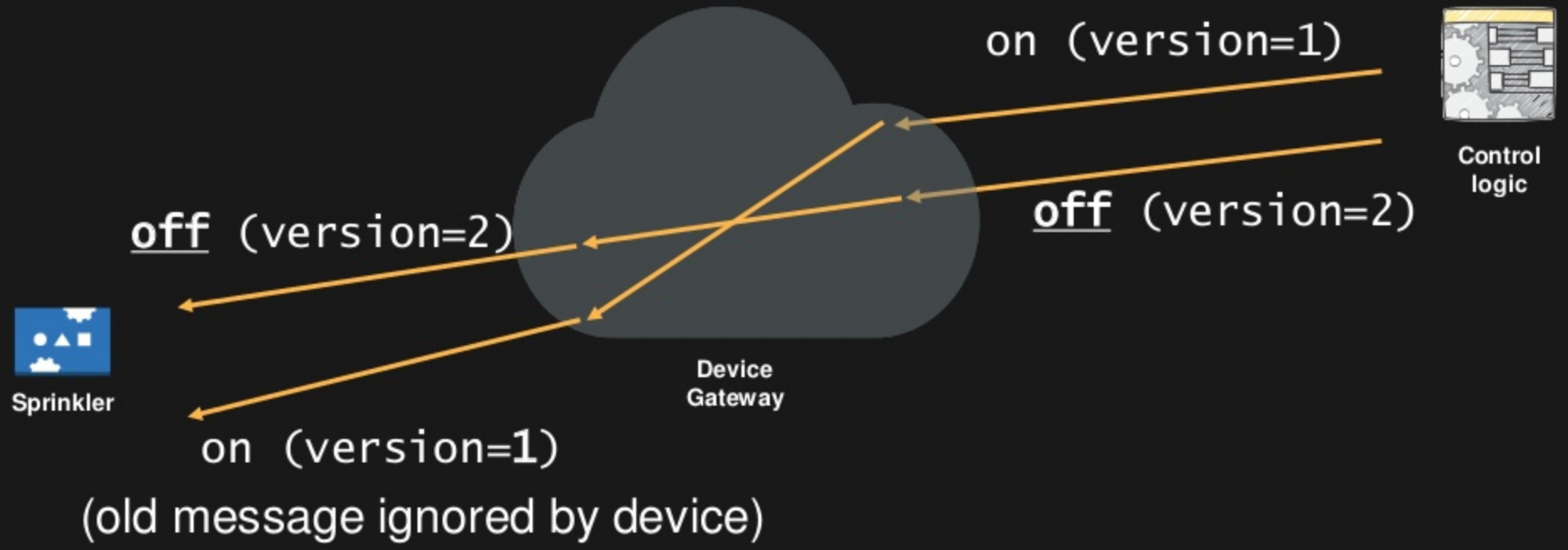
Shadow reports delta, desired and reported states along with metadata and version

Mobile App

Set the desired state of a device
Get the last reported state of the device
Delete the shadow

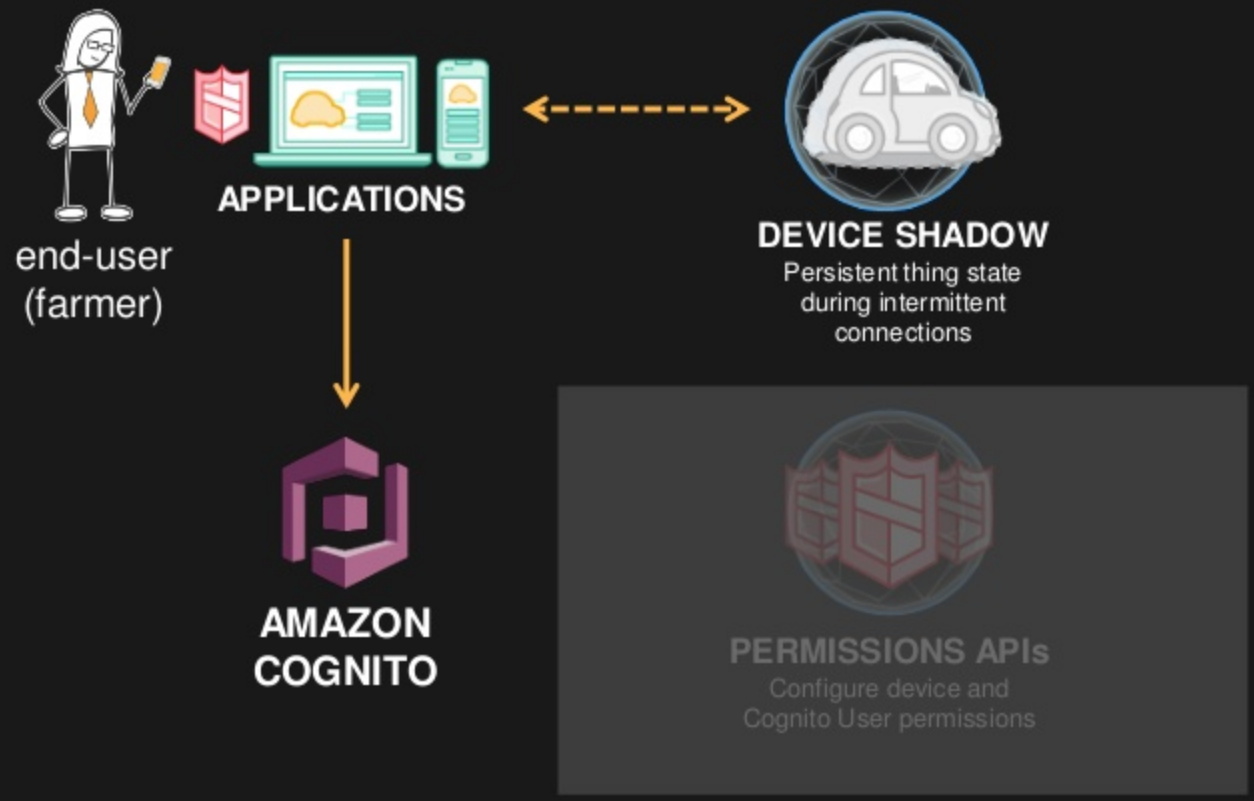
```
{
"state" : {
        "desired" : {
            "lights": { "color": "RED" },
            "engine" : "ON"
        },
        "reported" : {
            "lights" : { "color": "GREEN"  },
        "engine" : "ON"
        },
        "delta" : {
            "lights" : { "color": "RED"  }
         } },
"version" : 10
}
```

Device Shadows and versioning



Mobile Control

Using Amazon Cognito with IoT



Policy for Cognito with IoT

Cognito Identity = us-east-1:xxxx-yyyy-zzzz

```
aws iot attach-principal-policy --policy-name farm-sensors
--principal us-east-1:xxxx-yyyy-zzzz
```

You will need a trusted entity to attach the Cognito principal to an IoT policy

- Only needed for iot-data plane calls such as DeleteThingShadow, UpdateThingShadow, GetThingShadow, Connect, Publish, and Subscribe
- Can use API Gateway, Cognito Sync Triggers, or other techniques for attaching the Cognito principal ID to the IoT policy

Overall Cognito “pairing” workflow

1. Create a Cognito identity pool
2. Customer signs in using mobile app
3. Associate their user with their “farm”
4. Create a scope-down policy in IoT for their user
5. Attach that policy to their Cognito user in IoT

Managing fine-grained permissions

- One “farm owner” needs permissions to many shadows
  - "arn:aws:iot:…:thing/sprinkler123abc"
  - "arn:aws:iot:…:thing/sprinkler456def"
  - …
- Listing each is tedious

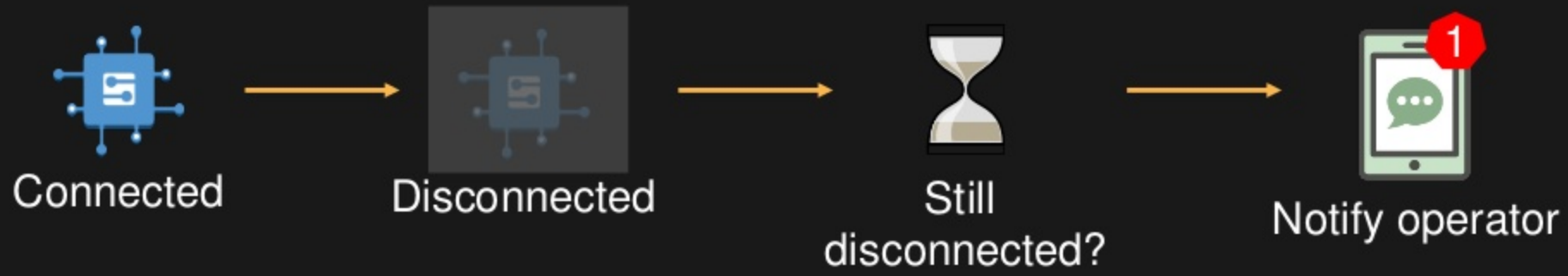
Best practice: thing name prefixing

- Prefix thing name with logical owner
  - `sensor123abc -> macdonald-sensor123abc`

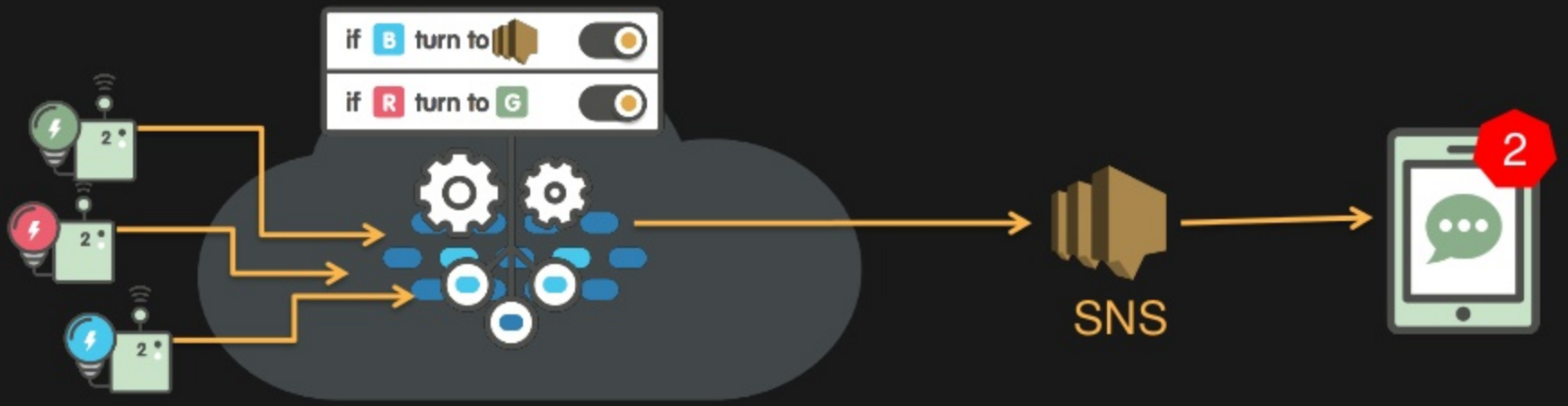
- IAM policies support wildcards
  - `"arn:aws:iot:…:thing/sensor123abc"`
  - `"arn:aws:iot:…:thing/sensor123abc"`
  - `"arn:aws:iot:…:thing/sensor456def"`
  - …
  - `"arn:aws:iot:…:thing/macdonald-*"`

Lifecycle Management

Lifecycle workflow



AWS IoT Rules Engine & Amazon SNS



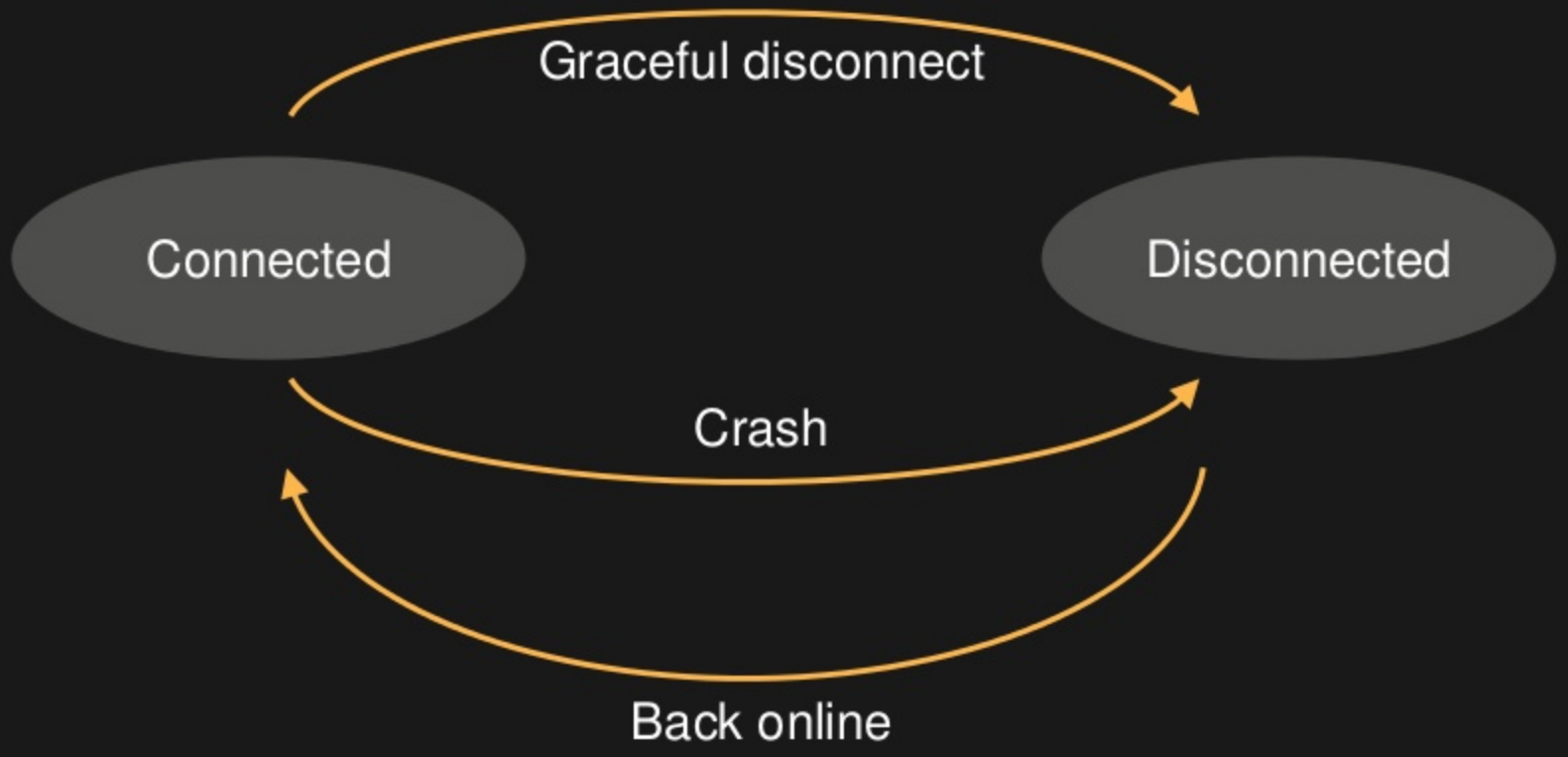
Push Notifications

Apple APNS Endpoint, Google GCM Endpoint, Amazon ADM Endpoint, Windows WNS

Amazon SNS -> HTTP Endpoint (or SMS or email)

Call HTTP-based third-party endpoints through SNS with subscription and retry support

Detecting disconnects



Lifecycle events

- Connect
  - `PUBLISH lifecycle/sensor-123`
    `{"status": "online"}`
- Disconnect (graceful)
  - `PUBLISH lifecycle/sensor-123`
    `{"status": "offline"}`
- Disconnect (crash)
  - `PUBLISH lifecycle/sensor-123`
    `{"status": "offline", "isCrash": true}`

Handling lifecycle events

```
SELECT
  status,
  topic(2) as deviceId,
  timestamp() as time,
  isCrash
FROM lifecycle/#
WHERE status='offline'
```

AWS Lambda function

- Look up mobile push ID for device owner
- Send SNS mobile push

Delayed lifecycle events

```
SELECT
  status,
  topic(2) as deviceId,
  timestamp() as time,
  isCrash
FROM lifecycle/#
```

AWS Lambda function

- Store update device status in DynamoDB
- If offline: enqueue an SQS message with DelaySeconds

SQS Message (15 minutes later)

- Double-check the status in DynamoDB
- Send SNS push notification if still offline

Amazon DynamoDB

Device	Status	Time
sensor-123	connected	11:30
...		

Generating lifecycle events

- Connect
  - `PUBLISH lifecycle/sensor-123`
    `{"status": "online"}`
- Disconnect (graceful)
  - `PUBLISH lifecycle/sensor-123`
    `{"status": "offline"}`
- Disconnect (crash)
  - `PUBLISH lifecycle/sensor-123`
    `{"status": "offline", "isCrash": true}`

Lifecycle events: connecting

Automatic lifecycle message

$aws/events/presence/connected/*clientId*


```
{
    "clientId": "a1b2c3d4e5f6a7b8c9d0e1f2a3b4c5d6",
    "timestamp": 1460065214626,
    "eventType": "connected",
    "sessionIdentifier": "00000000-0000-0000-0000-000000000000",
    "principalIdentifier": "000000000000/XXX:some-user/XXX:some-user"
}
```

Lifecycle events: disconnecting

Automatic lifecycle message

$aws/events/presence/disconnected/*clientId*

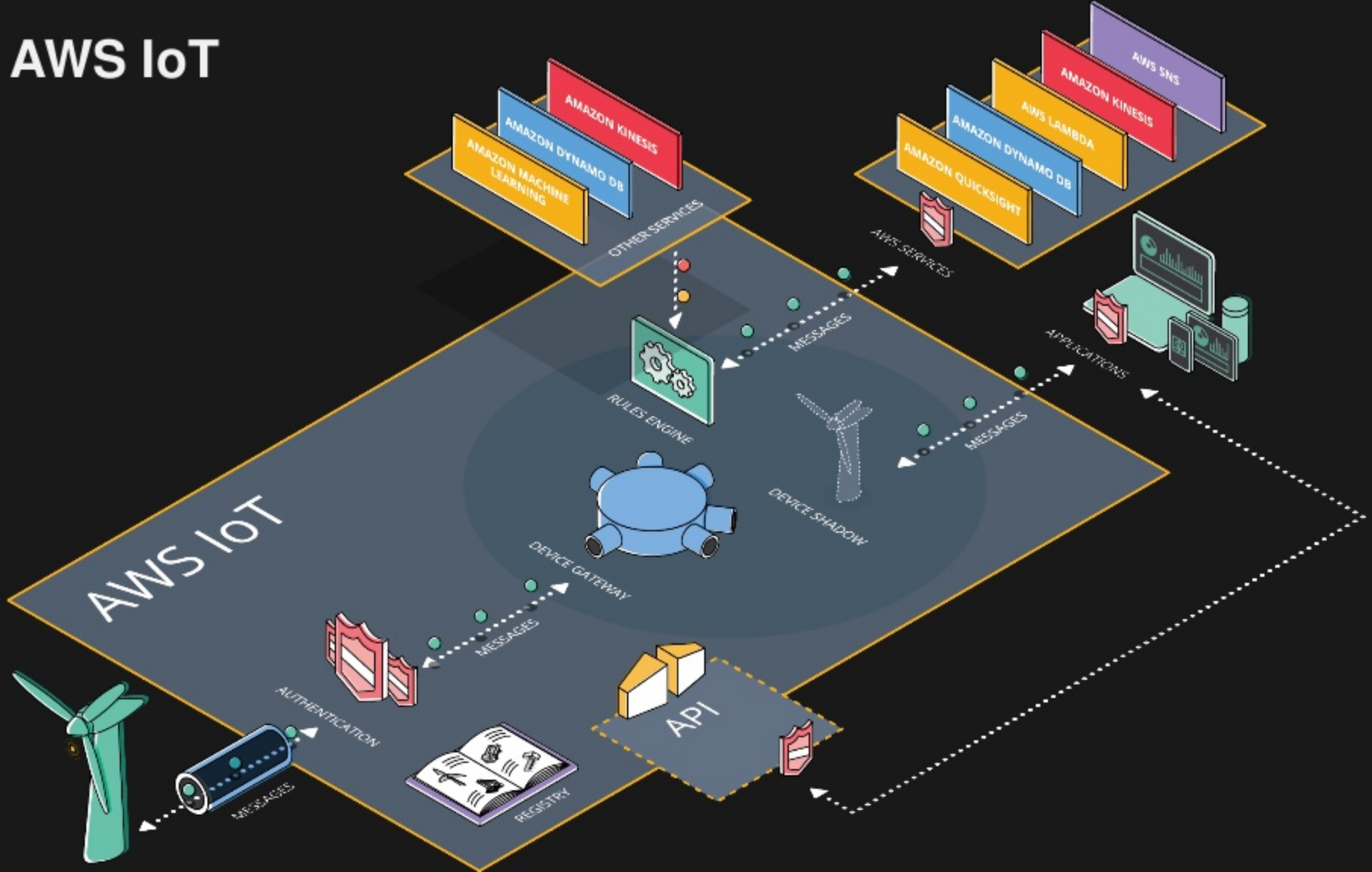

```
{
    "clientId": "a1b2c3d4e5f6a7b8c9d0e1f2a3b4c5d6",
    "timestamp": 1460065214626,
    "eventType": ”disconnected",
    "sessionIdentifier": "00000000-0000-0000-0000-000000000000",
    "principalIdentifier": "000000000000/XXX:some-user/XXX:some-user"
}
```

Last Will and Testament (LWT)

```
CONNECT message parts:
 Protocol: MQTT 3.1.1
 ClientId: abc
 KeepAlive: 60 seconds
 LastWill PUBLISH message:
    Topic: foo/bar
    QoS: 1
    Payload: {"foo": "bar"}
```

Wrap-up

AWS IoT


AMAZON KINESIS
AMAZON DYNAMO DB
AMAZON MACHINE LEARNING
OTHER SERVICES
AWS SNS
AMAZON KINESIS
AWS LAMBDA
AMAZON DYNAMO DB
AMAZON QUICKSIGHT
AWS SERVICES
MESSAGES
RULES ENGINE
APPLICATIONS
MESSAGES
DEVICE SHADOW
AWS IoT
DEVICE GATEWAY
MESSAGES
AUTHENTICATION
API
REGISTRY
MESSAGES

Key takeaways

- Messaging
  - Be careful with wide fan out
  - No message ordering guarantees
  - Avoid large fan in
  - WebSockets for Amazon Cognito authentication
- Rules
  - Send data to multiple data stores at the same time
  - Manage device lifecycle events
- Shadows
  - Designed for the real world: poor connectivity, out of order messages
  - Fine-grained control over software rollouts
  - Not ideal for storing time-series analytics data
- Security
  - One cert per device
  - Set fine-grained permissions for devices and Amazon Cognito users
  - Naming conventions can simplify policy management

AWS

SUMMIT

Thank you!

amazon web services