
AWS
re:Invent

SVS311

Handling errors in a serverless world

Josh Kahn

Principal Solutions Architect

AWS

AWS re:Invent

© 2020, Amazon Web Services, Inc. or its affiliates. All rights reserved.

```
{
    “message”: “Internal Server Error”
}
```

Session expectations

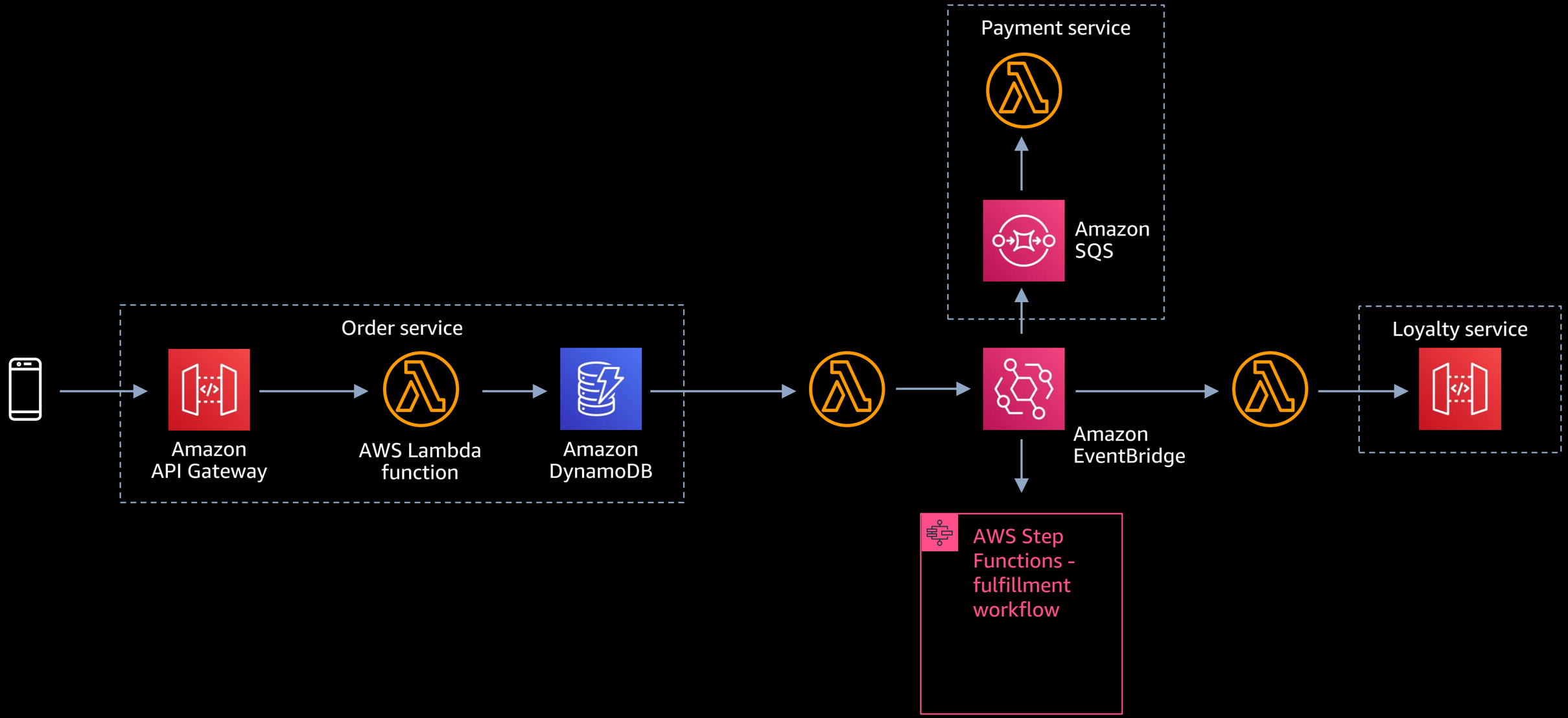
We will cover

- How to add reliability to serverless architectures by
  - Understanding built-in retry behaviors
  - Using configurations to modify those behaviors
- How to enhance visibility of errors
  - Across AWS Lambda invocation methods
  - While minimizing developer impact

We will not cover

- Observability
- Basics of serverless

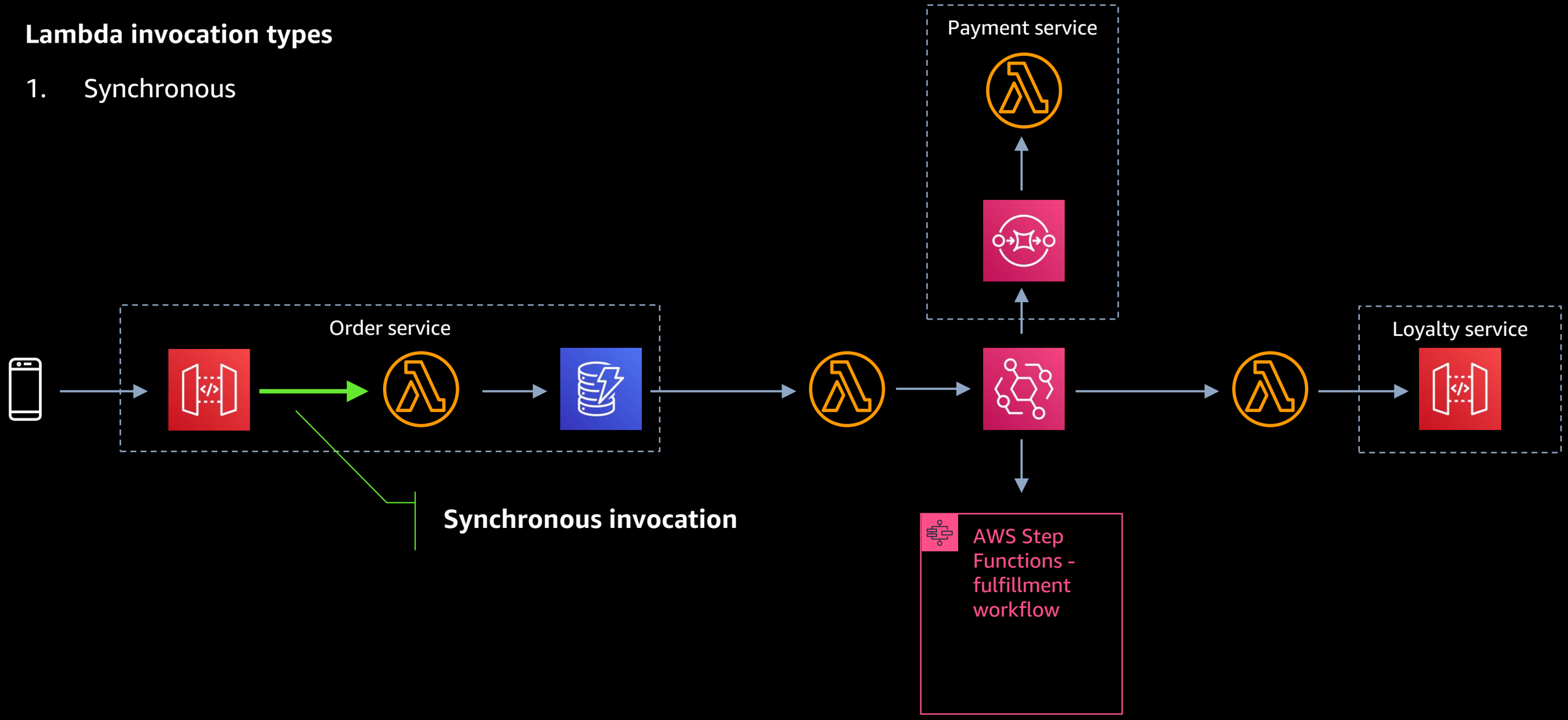
Serverless eCommerce architecture (simplified)



Serverless eCommerce architecture (simplified)

Lambda invocation types

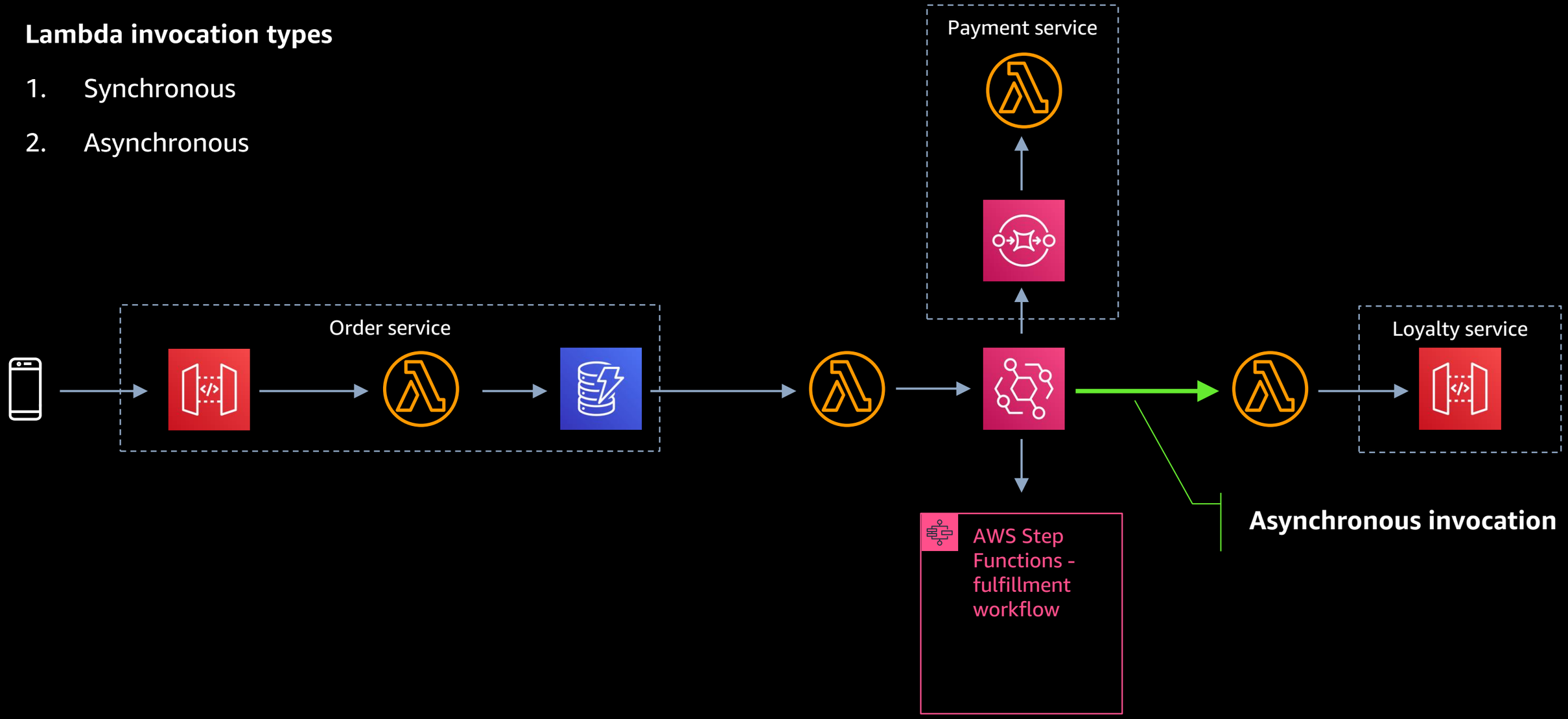
1. Synchronous



Serverless eCommerce architecture (simplified)

Lambda invocation types

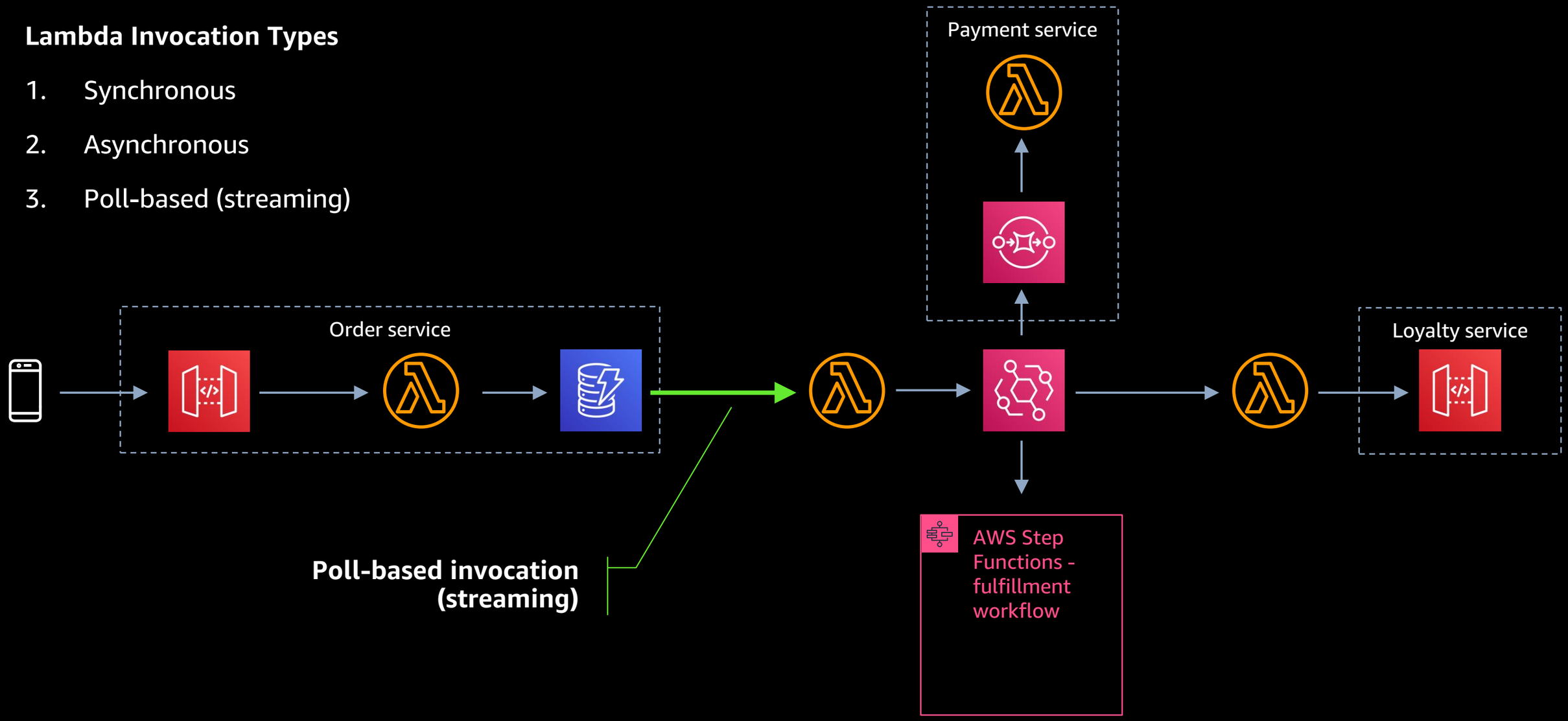
1. Synchronous
2. Asynchronous



Serverless eCommerce architecture (simplified)

Lambda Invocation Types

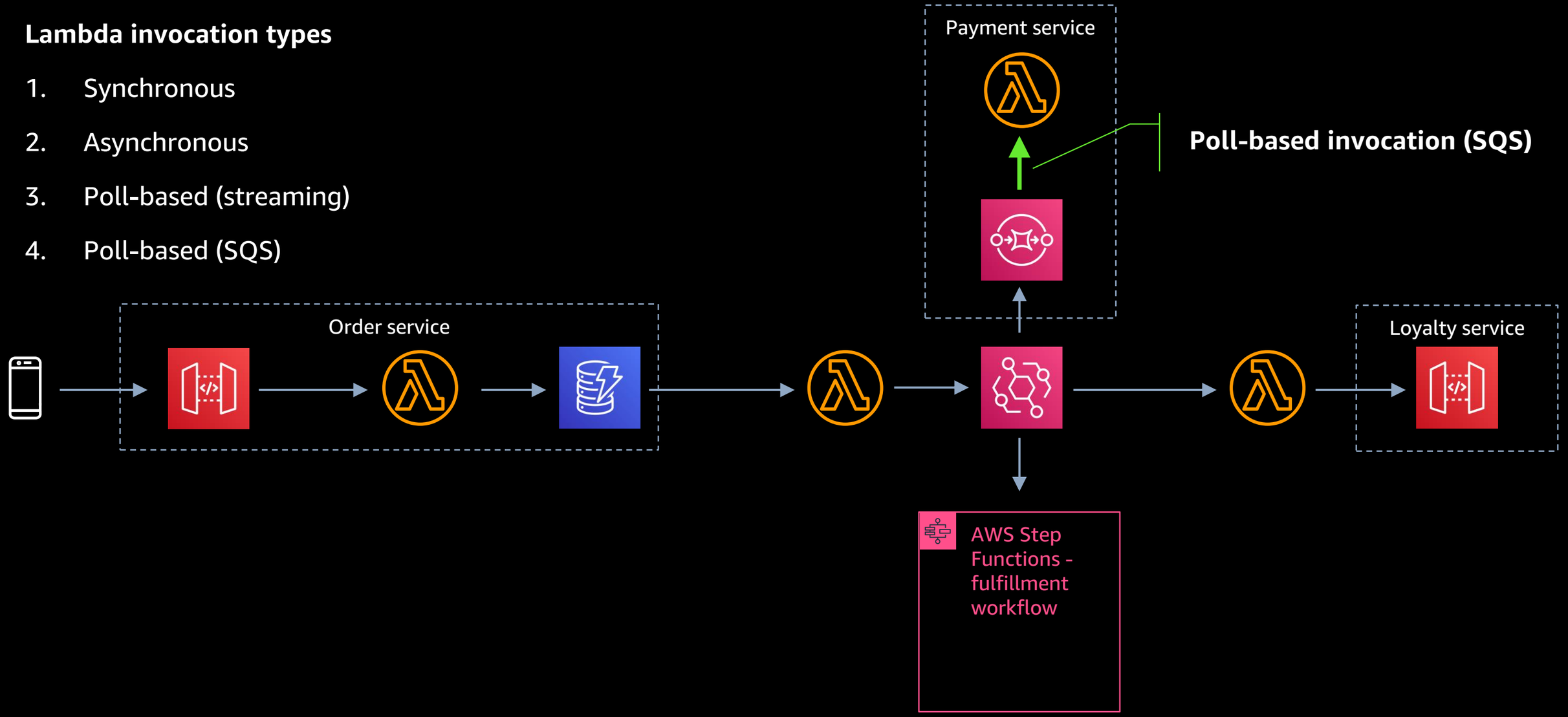
1. Synchronous
2. Asynchronous
3. Poll-based (streaming)



Serverless eCommerce architecture (simplified)

Lambda invocation types

1. Synchronous
2. Asynchronous
3. Poll-based (streaming)
4. Poll-based (SQS)



Synchronous invocations

Retry behavior: None

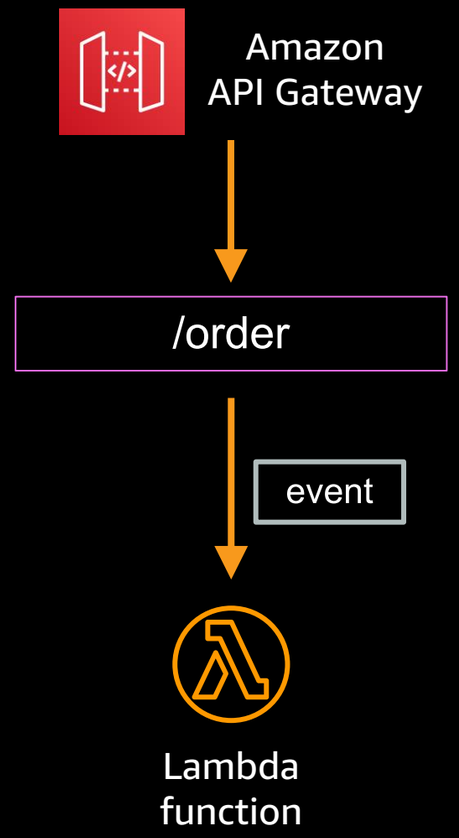
- Client must retry on failure

Goals

- Log errors to enable alarms, insights, error budgets
- Allow for tracing, API Gateway metrics
- Report sensible errors to caller

Error types

- Client (e.g., bad payload, record not found)
- Server / function (e.g., access denied, throttles)



Handling errors for synchronous invocations

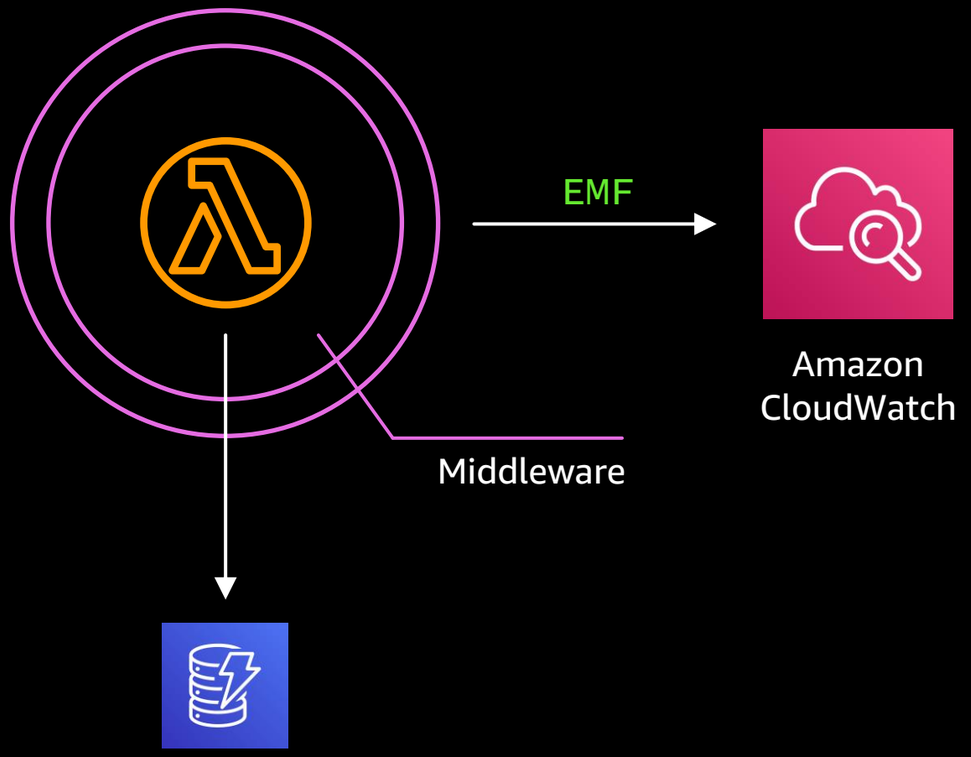
Lambda function

- `throw` custom client errors (400)
  - e.g., `BookNotFoundError`
- Do **not** `catch` function errors (500)
  - e.g., `AccessDeniedError`

Middleware

- Catch all errors / exceptions
- Log errors to Amazon CloudWatch using embedded metric format (EMF) with Lambda context

- If custom error, return success with sanitized error message and code (e.g., 404)
- Else, **fail** with sanitized error



Sample sync middleware implementation



```python
@error_handler
def handler(event, context):
  book = find_book(event.get('book_id'))
  if book is None:
    raise BookNotFoundError(book_id) # custom exception

  return { "statusCode": 200, "body": json.dumps(book) }

@lambda_handler_decorator(trace_execution=True) # aws_lambda_powertools middleware
def error_handler(handler, event, context):
  logger = Logger(child=True)

  try:
    return handler(event, context)
  except Exception as err:
    logger.exception(err) # log error message and metric
    if err.code < 500:
      return sanitize_error(err) # return sanitized error to client

    raise CustomException from err # re-raise error to indicate failure
```

Capture errors in AWS X-Ray tracing

Raise error from function to

- Easily identify errors in traces
- Create error budgets via X-Ray trace groups

In Node.js (async handler):

```
if (error.statusCode >= 500) {
  return Promise.reject(error);
} else {
   return Promise.resolve(error);
}
```

AWS X-Ray



Asynchronous invocations

Retry behavior – 0-2 additional attempts

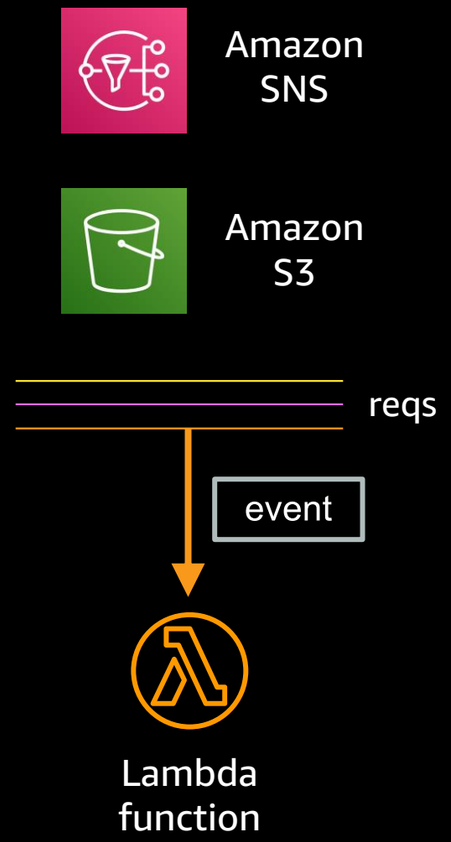
- Throttled / system errors – hours to days
- Charged as additional invocations

Goals

- Log errors to enable alarms, insights
- Allow for replay, review of failing events

Configuration

- `MaximumEventAge`
- `MaximumRetryAttempts`



Brief review of Lambda destinations

- Route asynchronous function results to a destination
- Route by execution status: `onSuccess` or `onFailure`
- Includes metadata on failed events
  - Retry attempts
  - Function name
- No additional code

```
# Configure async Lambda Destination with AWS SAM
MyAsyncFunction:
  Type: AWS::Serverless::Function
  Properties:
    Runtime: python3.8
    Handler: app.lambda_handler
    ...
    EventInvokeConfig:
      DestinationConfig:
        OnSuccess:
          Type: Lambda
          Destination: !GetAtt DestinationLambda.Arn
        OnFailure:
          Type: SQS
          Destination: !GetAtt MyFailureQueue.Arn
```


Amazon SNS


Amazon SQS


Amazon EventBridge


AWS Lambda

Handling errors for asynchronous invocations

Lambda function

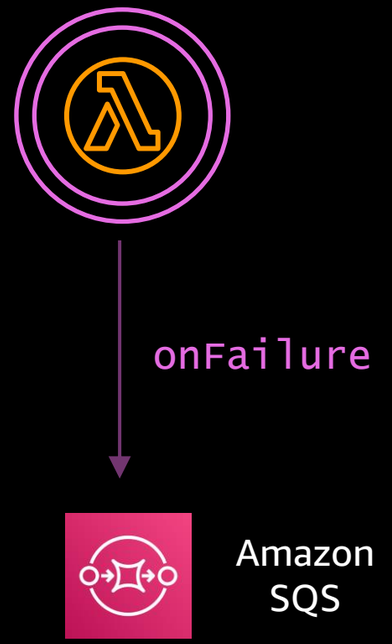
- If function calls another service
  - Consider retrying or circuit breaker pattern
- `throw` errors

Middleware

- Catch all errors
- Log errors with metadata using CloudWatch EMF
- Re-raise errors to send to `onFailure` destination

`onFailure` destination

- Destination payload includes original request payload and associated metadata



Sample async middleware implementation



```
@error_handler
def handler(event, context):
    return { ... }

@lambda_handler_decorator(trace_execution=True) # aws_lambda_powertools middleware
def error_handler(handler, event, context):
  logger = Logger(child=True)
  handler = logger.inject_lambda_context(handler) # inject Lambda context info

  try:
    return handler(event, context)
  except botocore.exceptions.ClientError as err:
    logger.exception(err.response) # log error message and metric for AWS clients
    raise CustomException from err
  except Exception as err:
    logger.exception(err) # log error message and metric
    raise CustomException from err
```

Streaming event sources

Retry behavior: Until message expires

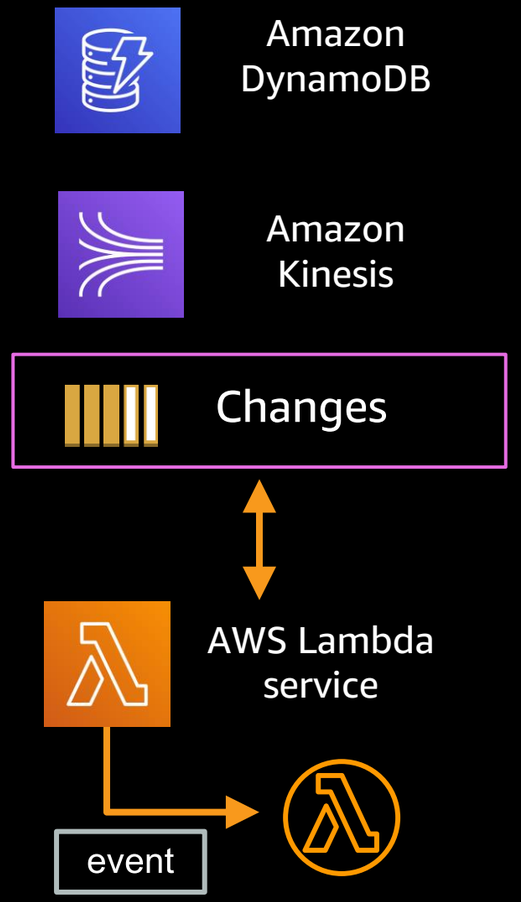
- Retries entire batch of messages

Goals

- Log errors to enable alarms, insights
- Allow for replay, review of failing events

Configuration

- `MaximumRecordAge`
- `MaximumRetryAttempts`
- `BisectBatchOnFunctionError`



Handling errors for streaming event sources

Lambda function

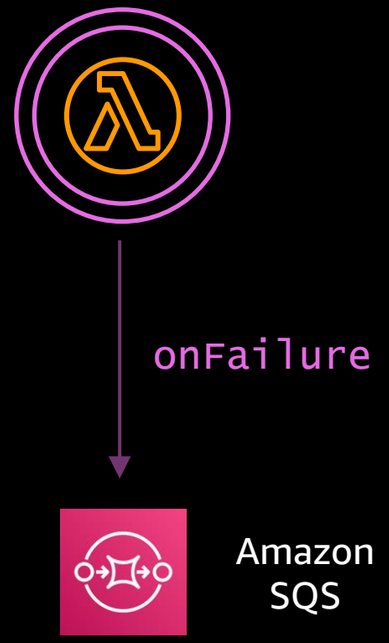
- `throw` errors

Middleware

- Log errors with metadata using CloudWatch EMF
- Re-raise error
- If function is idempotent, use `BisectBatchOnFunctionError` to retry failed batch
  - Increases function invocations, but not counted toward retry limit

`onFailure` destination

- Receives stream metadata (not the record)
- Use to retrieve original record



Lambda destinations for streaming sources

```
MyStreamingFunction:
  Type: AWS::Serverless::Function
  Properties:
    ...
 Events:
   Stream:
     Type: DynamoDB
     Properties:   # Destination and retry behavior are properties of stream
       Stream: !GetAtt SampleDDBTable.StreamArn
       StartingPosition: TRIM_HORIZON
       MaximumRecordAgeInSeconds: 60
       MaximumRetryAttempts: 1
       DestinationConfig:
         OnFailure:
           Type: SQS
           Destination: !GetAtt FailureDestinationQueue.Arn
```



Amazon SNS



Amazon SQS

Amazon SQS event sources

Retry behavior: Until message expires

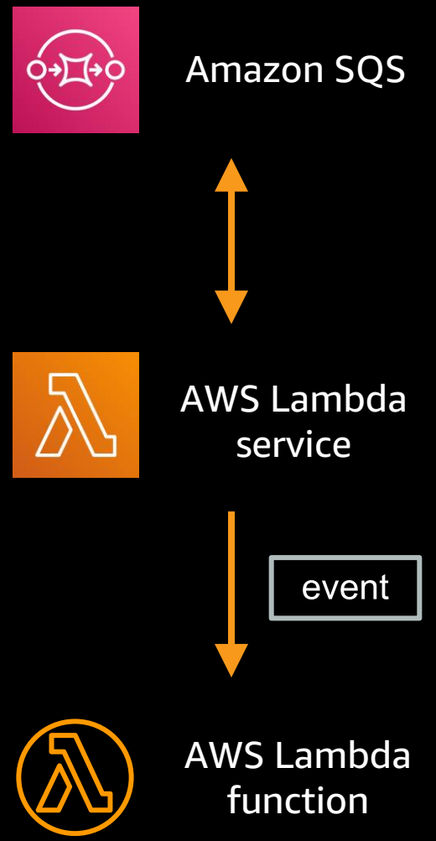
- Default – retry all messages in batch
- Function can delete completed messages

Goals

- Log errors to enable alarms, insights
- Handle batches with partial failures

Configuration

- Dead-letter queue (Lambda and Amazon SQS)
- Batch size (up to 10)
- Visibility timeout



Handling errors for Amazon SQS event sources

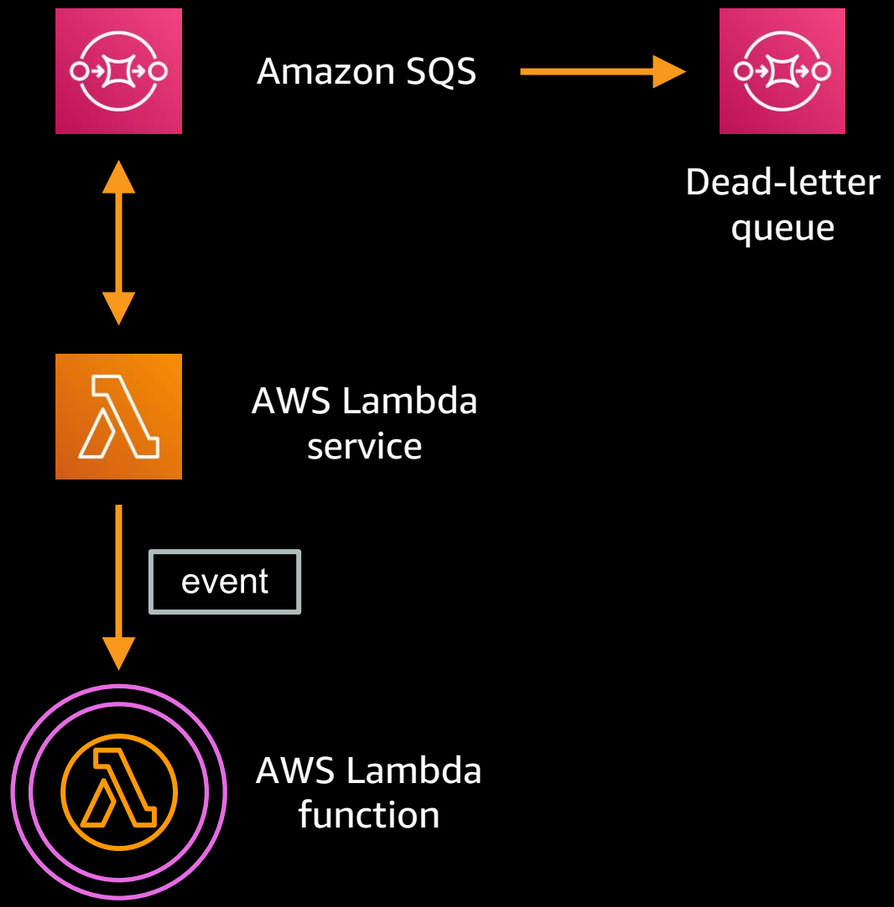
Lambda function

- Iterate through each record in batch
- `catch` errors while processing each record

Middleware

- If no errors, `return`
  - Lambda service poller will delete messages
- Else
  - Delete successful messages (`DeleteMessage`)
  - Raise aggregate error to return failed messages to queue

Use dead-letter queue on Amazon SQS



Handling partial batch failures in SQS sources



```python
from aws_lambda_powertools.utilities.batch import sqs_batch_processor

def record_handler(record):
  # Called for each message in the batch
  # Raise an exception if message processing fails
  return_value = do_something_with(record["body"])

  if error:
    raise error

  return_value

@sqs_batch_processor(record_handler = record_handler) # aws_lambda_powertools middleware
def lambda_handler(event, context):
  return { "statusCode": 200 }
```

Step Functions event sources

Retry behavior: Configurable

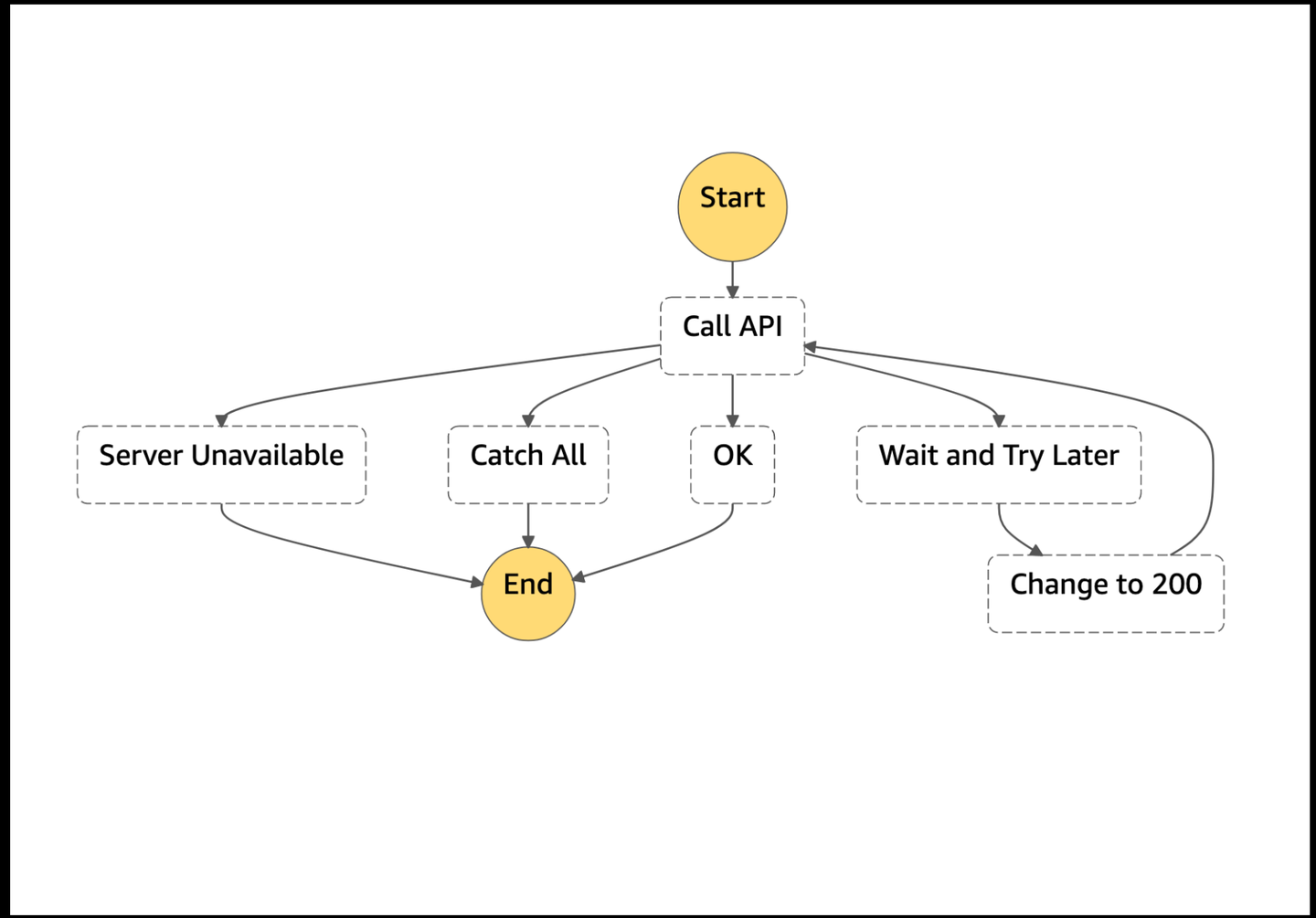
- Specify using `Retry`, by error type

Goals

- Centralize error handling logic in state machine
- Log errors to enable alarms, insights

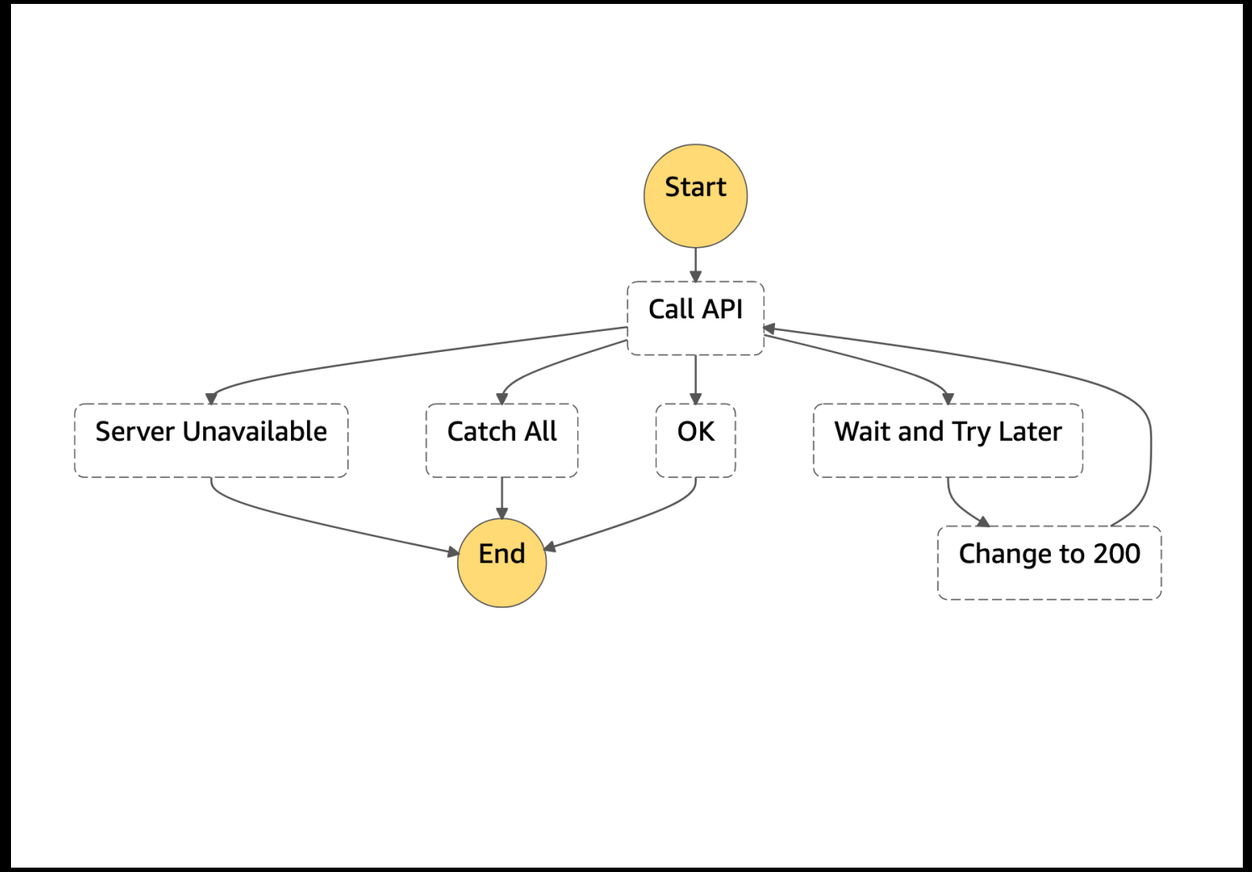
Configuration

- Support for retry and try / catch in Amazon States Language
- Customizable to need, error type



Handling errors in state machines

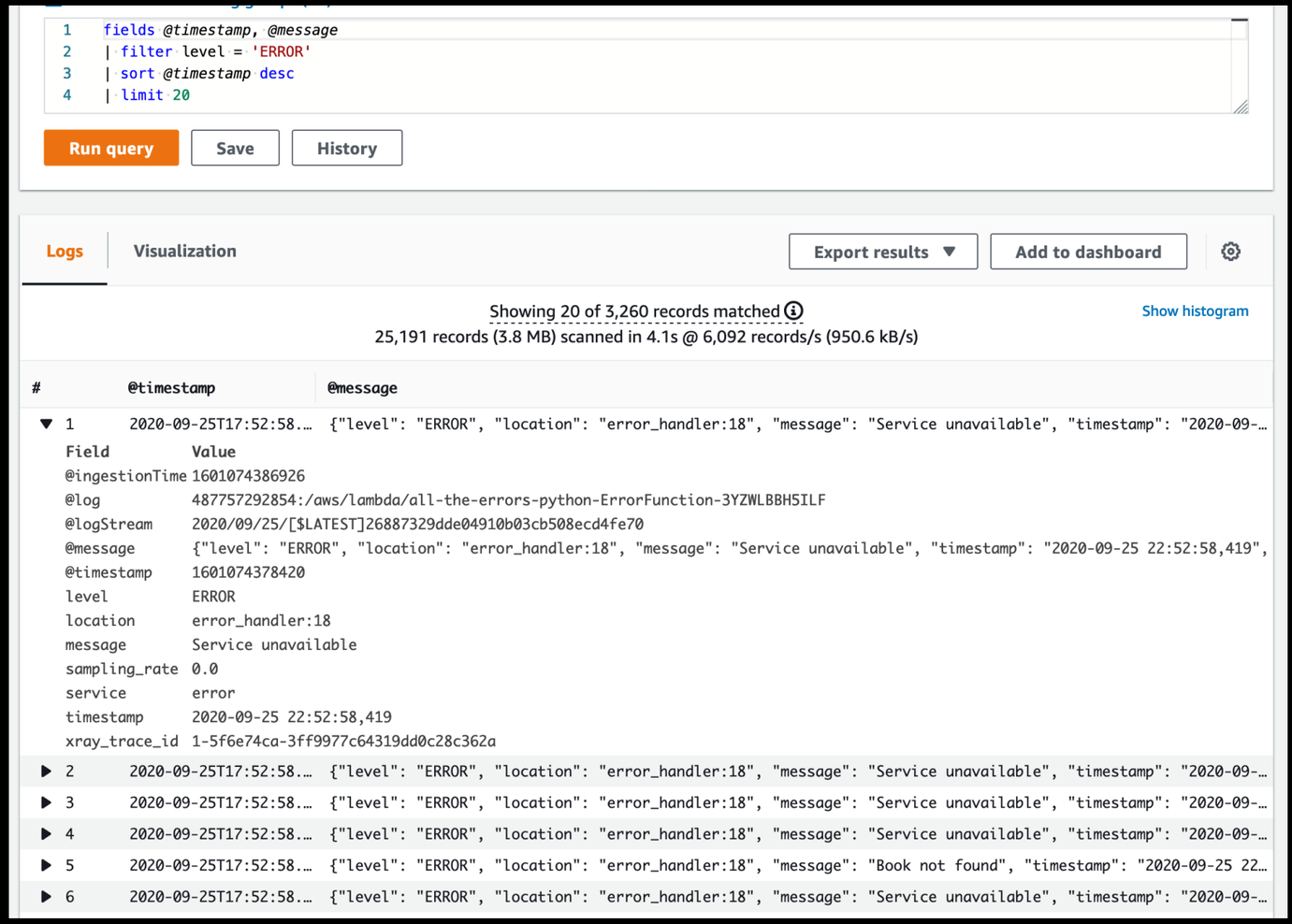
```
"Call API": {
    "Type": "Task",
    "Resource": "arn:aws:lambda:us-east-2:1234567890_ID:function:my_lambda_function",
    "Next": "OK",
    "Retry": [{
        "ErrorEquals": ["TooManyRequestsException"],
        "IntervalSeconds": 1,
        "MaxAttempts": 2
    }],
    "Catch": [{
        "ErrorEquals": ["TooManyRequestsException"],
        "Next": "Wait and Try Later"
    }, {
        "ErrorEquals": ["ServerUnavailableException"],
        "Next": "Server Unavailable"
    }, {
        "ErrorEquals": ["States.ALL"],
        "Next": "Catch All"
    }]
}
```



Error visibility

- ✓ CloudWatch embedded metric format (EMF)
- ✓ CloudWatch Logs Insights
- ✓ AWS X-Ray (where available)

- Other opportunities
  - CloudWatch dashboards
  - CloudWatch alarms
  - CloudWatch Lambda Insights (preview)
  - Chaos engineering



Thank you!

Josh Kahn

Principal Solutions Architect
AWS

Session resources:
https://s12d.com/NTuMteDa



AWS re:Invent

© 2020, Amazon Web Services, Inc. or its affiliates. All rights reserved.



Please complete the session survey

AWS re:Invent

© 2020, Amazon Web Services, Inc. or its affiliates. All rights reserved.