
AWS
re:Invent
DECEMBER 2 – 6, 2024 | LAS VEGAS, NV

OPN402

Gain expert-level knowledge about Powertools for AWS Lambda

Andrea Amorosi
(he/him)
Senior SA Engineer, Powertools
Amazon Web Services

01 Powertools for AWS Lambda

02 Structured logging

03 Event handler

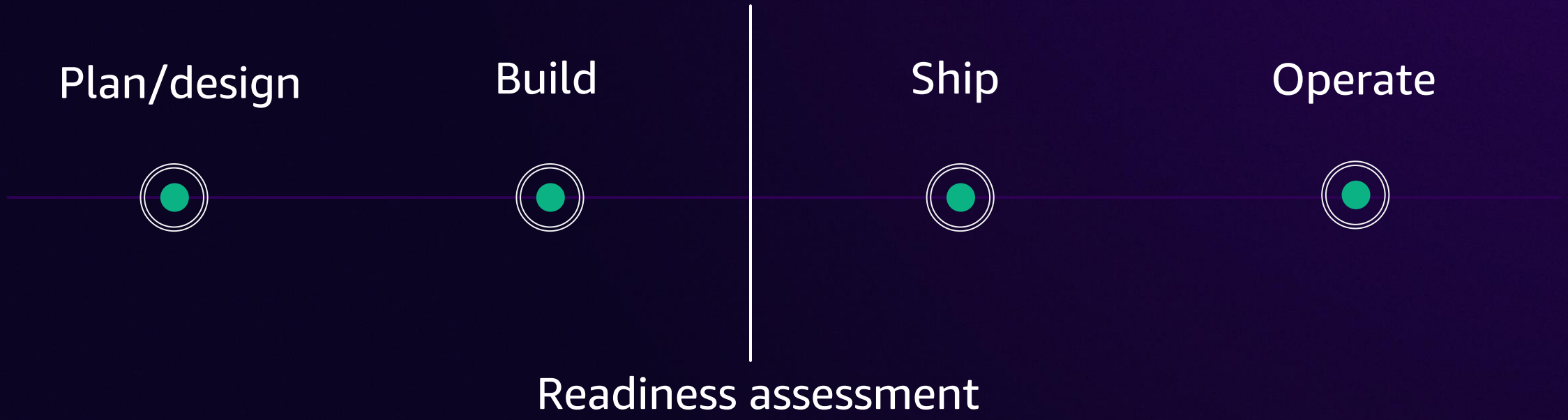
04 Idempotency

05 Batch processing

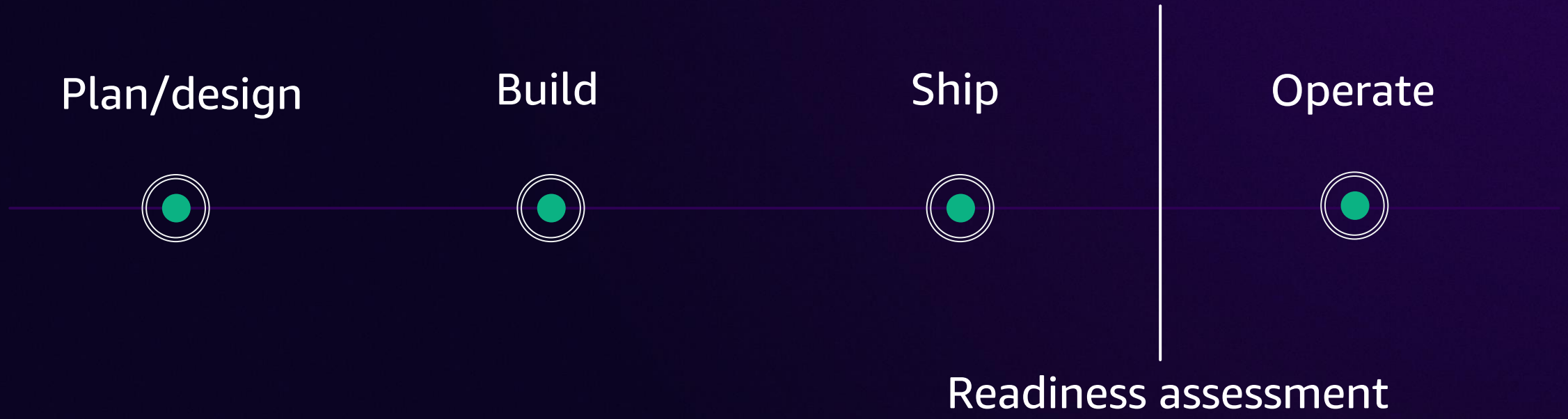
06 Roadmap and wrap up

Systems development life cycle

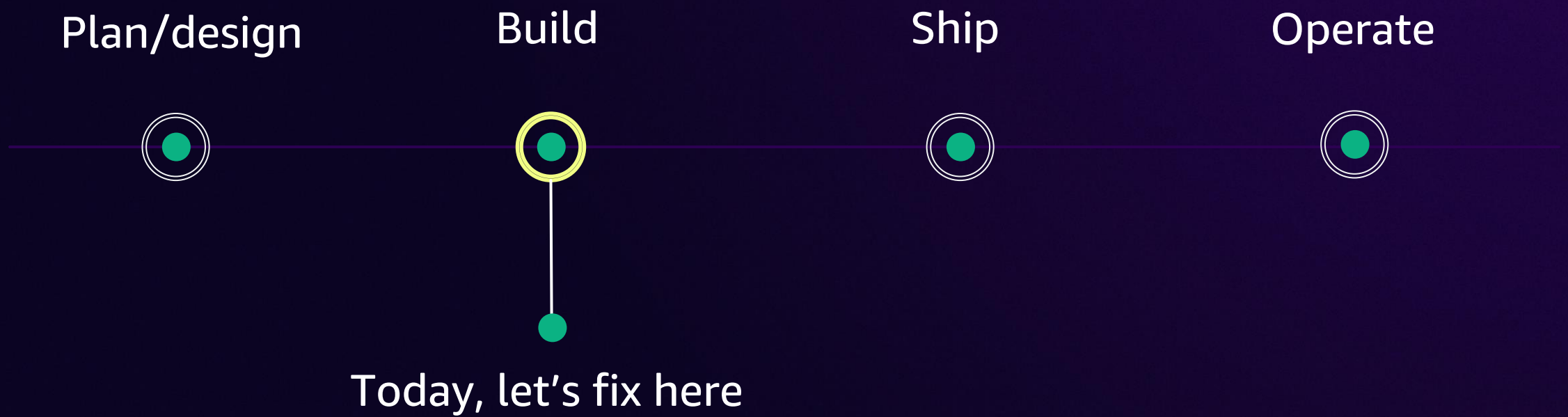
Findings come in late



Findings come in late



Let’s fix this



Powertools for AWS Lambda

- Logger
- Metrics
- Tracer
- + Many more

A toolkit to implement serverless **best practices** and increase **developer velocity**

Powertools for AWS Lambda: Toolkit

Best practices for **everyone**

Python | TypeScript | Java | .NET

- Batch processing
- Config management
- Observability
- Secrets handling
- REST/GraphQL API
- Idempotency
- Input/output validation
- BYO middleware
- Self-documented schemas
- Feature flags
- Caching
- Streaming
- Data extraction

*feature set may vary across languages

Powertools for AWS Lambda

200B+
requests/week

40%
Community developed

Structured logging

Structured logging: Recap

Raw Semi-structured Canonical Structured

Structured logging: Without Powertools

Raw

```python
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)

logger.info("Hello world")
```

[INFO] 2024-12-02T22:.. 1c8df7d3... Hello world

Structured logging: Without Powertools

Semi-structured

```python
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)

logger.info({"message": "Hello world"})
```

[INFO] 2024-12-02T22:.. … {“message”: “Hello world”}

Structured logging: Without Powertools

Canonical

```python
import logging
import os

from logfmter import Logfmter

handler = logging.StreamHandler()
handler.setFormatter(Logfmter())
logging.basicConfig(handlers=[handler], level=os.getenv("LOG_LEVEL", "INFO"))

logging.info("Aha!", extra={"request_latency": 0.1})

```

at=INFO msg=Aha! request_latency=0.1

Structured logging: Without Powertools

Structured

```python
import logging
import os

from pythonjsonlogger import jsonlogger

logger = logging.getLogger()
structured_handler = logging.StreamHandler()
formatter = jsonlogger.JsonFormatter(
    fmt='%(asctime)s %(levelname)s %(name)s %(message)'
)

structured_handler.setFormatter(formatter)
logger.addHandler(structured_handler)
logger.setLevel(os.getenv("LOG_LEVEL", "INFO"))

logger.info("Aha!")

```

Structured logging: Without Powertools

Structured

```python
import logging
import os

from pythonjsonlogger import jsonlogger

logger = logging.getLogger()
structured_handler = logging.StreamHandler()
formatter = jsonlogger.JsonFormatter(
    fmt='%(asctime)s %(levelname)s %(name)s %(message)'
)

structured_handler.setFormatter(formatter)
logger.addHandler(structured_handler)
logger.setLevel(os.getenv("LOG_LEVEL", "INFO"))

logger.info("Aha!")

```

```json
{
  "asctime": "2024-12-02...",
  "levelname": "INFO",
  "name": "root",
  "message": "Aha!"
}
```

Structured logging: Powertools

Structured

```python
from aws_lambda_powertools import Logger

logger = Logger(service="payment")

logger.info("Aha!")
```

Structured logging: Powertools

Structured

```python
from aws_lambda_powertools import Logger

logger = Logger(service="payment")

logger.info("Aha!")
```

```json
{
  "level": "INFO",
  "location": "main.handler:5",
  "message": "Aha!",
  "timestamp": "2024-12-02...",
  "service": "payment",
  "sampling_rate": 0.0
}
```

Structured logging: Powertools

Structured

```python
from aws_lambda_powertools import Logger
from aws_lambda_powertools.logging import correlation_paths

logger = Logger(service="payment")

@logger.inject_lambda_context(correlation_id_path=correlation_paths.API_GATEWAY_REST)
def handler(event, context):
    logger.info("Aha!")

```

Structured logging: Powertools

Structured

```python
from aws_lambda_powertools import Logger
from aws_lambda_powertools.logging import correlation_paths

logger = Logger(service="payment")

@logger.inject_lambda_context(correlation_id_path=correlation_paths.API_GATEWAY_REST)
def handler(event, context):
    logger.info("Aha!")

```

```json
{
  "level": "INFO",
  "location": "main.handler:8",
  "message": "Aha!",
  "timestamp": "2024-12-02 11:47:12",
  "service": "payment",
  "cold_start": true,
  "lambda_function_name": "test",
  "lambda_function_memory_size": 128,
  "lambda_function_arn": "arn:aws:lambda:eu-west-1:123456789012:function:test",
  "lambda_request_id": "52fdfc07-2182-154f-164f-5f0f9a621d84",
  "correlation_id": "5a2f91db-c398-4afe-8d89-089df20c5cdc"
}
```

Structured logging: Powertools

BYO formatter

```python
from aws_lambda_powertools import Logger
from aws_lambda_powertools.logging.formatter import LambdaPowertoolsFormatter
from aws_lambda_powertools.logging.types import LogRecord


class CustomFormatter(LambdaPowertoolsFormatter):
  def serialize(self, log: LogRecord) -> str:
    return self.json_serializer({
      "level": log["level"],
      "message": log["message"],
      "timestamp": log["timestamp"],
      "correlation_ids": {
        "api_request_id": log["correlation_id"],
        "service": log["service"],
        "fn_request_id": log["lambda_request_id"]
      },
      "function_metadata": {
        "name": log["lambda_function_name"],
        "memory": log["lambda_function_memory_size"],
        "arn": log["lambda_function_arn"],
        "cold_start": log["cold_start"]
      }
    })


logger = Logger(service="payment", logger_formatter=CustomFormatter())
logger.info("hello")
```

Structured logging: Powertools

```
1 { "level": "INFO", "message": "validated user token", "timestamp": "2021-05-3 11:47:12", "customer_id": "868a2c8ef", ... }
2 { "level": "INFO", "message": "updated subscription", "timestamp": "2021-05-3 11:47:30", "customer_id": "868a2c8ef", ... }
3 { "level": "INFO", "message": "email notification ok", "timestamp": "2021-05-3 11:47:30", "customer_id": "868a2c8ef", ... }
```

Structured logging: Powertools

Wide logs

```python
from aws_lambda_powertools import Logger

logger = Logger(service="payment")

@logger.inject_lambda_context()
def handler(event, context):
  customer = validate_user_token(event)
  logger.append_keys(customer_id=customer.id)

  try:
    subscription_id = update_subscription(customer_id=customer.id)
    logger.append_keys(subscription_id=subscription_id)
  except Exception as exc:
    logger.append_keys(subscription_update_fault=exc)

  notification_id = send_email_notification(email=customer.email)
  logger.append_keys(notification_id=notification_id)

  logger.info('subscription process')
```

Structured logging: Powertools

Wide logs

```
{
  "level": "INFO",
  "message": "subscription process",
  "timestamp": "2024-12-02 11:47:12",
  "customer_id": "868a2c8ef",
  "subscription_id": "4f5ccf70f56c4998b191ba3af7bf3783",
  "notification_id": "0b478d8ca6fc43388ba6",
  "correlation_ids": {
    "service": "payment",
    "fn_request_id": "addc193-497c-a446-290d-16002cb6b134a"
  },
  "function_metadata": {
    "name": "test",
    "memory": 128,
    "arn": "arn:aws:lambda:eu-west-1:123456789012:function:test",
    "cold_start": false
  }
}
```

Event handler

A common pattern

Routing

Input/output validation

Observability

Serialization

Idempotency

OpenAPI

Swagger UI

Config management

BYO middleware

API design & trade-offs

How many Lambda functions do you need for an API?

API design & trade-offs

Lambda monolith

?

Micro-functions

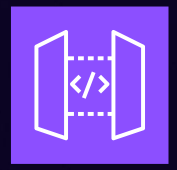
Lambda monolith

/orders

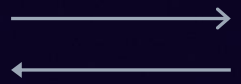
- Simplicity
- Lower cold start chance
- Higher cold start time
- Scaling & quotas
- Broad permissions
- Simpler CI/CD

Micro-functions

/orders



Amazon API Gateway



GetOrderFunction

ChangeOrderFunction

CancelOrderFunction

- Complexity
- Higher cold start chance
- Lower cold start time
- Independent scaling
- Granular permissions
- Multiple deployments

Event handler

Understand input structure

Create routing

Extract and transform input

Create response structure

Extract path and query parameters

Handle exceptions

```python
def handler(event, contex):
  try:
    http_method = event["httpMethod"]
    if http_method == "POST":
      payload = json.loads(event["body"])
      order_id = create_order(payload)
      if not order_id:
        return {
          "statusCode": 500,
          "body": json.dumps({"message": "Could not create order"})
        }
      return {
        "statusCode": 201,
        "body": json.dumps({"order_id": order_id})
      }
    if http_method == "GET":
      order_id = event["pathParameters"]["order_id"]
      resp = get_order(order_id)
      return {
        "statusCode": 200,
        "body": json.dumps(resp)
      }
  except:
    return {
      "statusCode": 500,
      "body": json.dumps({"message": "Internal Server Error"})
    }

```

Routing

```python
from aws_lambda_powertools.event_handler.api_gateway import APIGatewayRestResolver
from aws_lambda_powertools.utilities.typing import LambdaContext
import requests

app = APIGatewayRestResolver()


@app.get("/todos/<todo_id>")
def get_todo_by_id(todo_id: int):
    todo = requests.get(f"https://jsonplaceholder.typicode.com/todos/{todo_id}")
    todo.raise_for_status()

    return todo.json()


@app.route("/todos", methods=["POST", "PUT"])
def create_todo():
    todo_data: dict = app.current_event.json_body  # deserialize json str to dict
    todo = requests.post("https://jsonplaceholder.typicode.com/todos", data=todo_data)
    todo.raise_for_status()

    return todo.json()


def lambda_handler(event: dict, context: LambdaContext):
    return app.resolve(event, context)

```

Exception handling

```python
@app.exception_handler(ValueError)
def handle_invalid_limit_qs(ex: ValueError):  # receives exception raised
    metadata = {"path": app.current_event.path, "query_strings": app.current_event.query_string_parameters}
    logger.error(f"Malformed request: {ex}", extra=metadata)

    return Response(
        status_code=400,
        content_type=content_types.TEXT_PLAIN,
        body="Invalid request parameters.",
    )


@app.get("/todos")
@tracer.capture_method
def get_todos():
    # educational purpose only: we should receive a `ValueError`
    # if a query string value for `limit` cannot be coerced to int
    max_results: int = int(app.current_event.get_query_string_value(name="limit", default_value=0))

    todos: requests.Response = requests.get(f"https://jsonplaceholder.typicode.com/todos?limit={max_results}")
    todos.raise_for_status()

    return {"todos": todos.json()}
```

Validation

```python
from typing import Optional

import requests
from pydantic import BaseModel, Field

from aws_lambda_powertools.event_handler import APIGatewayRestResolver
from aws_lambda_powertools.utilities.typing import LambdaContext

app = APIGatewayRestResolver(enable_validation=True)


class Todo(BaseModel):
    userId: int
    id_: Optional[int] = Field(alias="id", default=None)
    title: str
    completed: bool


@app.get("/todos/<todo_id>")
def get_todo_by_id(todo_id: int) -> Todo:
    todo = requests.get(f"https://jsonplaceholder.typicode.com/todos/{todo_id}")
    todo.raise_for_status()

    return todo.json()

def lambda_handler(event: dict, context: LambdaContext) -> dict:
    return app.resolve(event, context)
```

OpenAPI spec

```python
import requests

from aws_lambda_powertools.event_handler import APIGatewayRestResolver
from aws_lambda_powertools.utilities.typing import LambdaContext

app = APIGatewayRestResolver(enable_validation=True)


@app.get(
    "/todos/<todo_id>",
    summary="Retrieves a todo item",
    description="Loads a todo item identified by the `todo_id`",
    response_description="The todo object",
    responses={
        200: {"description": "Todo item found"},
        404: {
            "description": "Item not found",
        },
    },
    tags=["Todos"],
)
def get_todo_title(todo_id: int) -> str:
    todo = requests.get(f"https://jsonplaceholder.typicode.com/todos/{todo_id}")
    todo.raise_for_status()

    return todo.json()["title"]
```

And more...

OpenAPI spec

Middlewares

Swagger UI

CORS

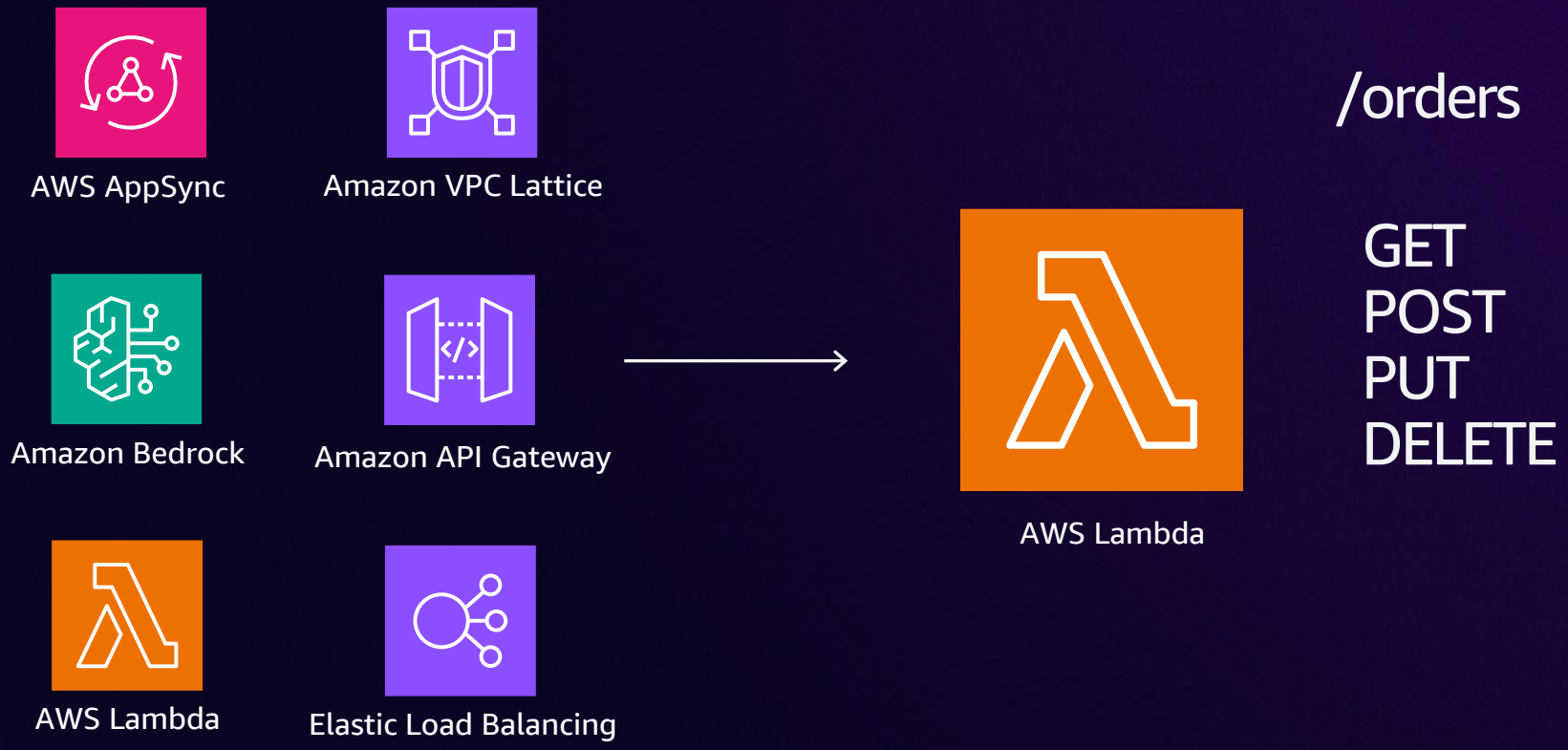
Fine-grained responses

Custom serializer

Binary repsonse

Compression

Event handler



Event handler

```python
import requests
from requests import Response

from aws_lambda_powertools.event_handler import APIGatewayRestResolver
from aws_lambda_powertools.utilities.typing import LambdaContext

app = APIGatewayRestResolver()


@app.get("/todos")
@tracer.capture_method
def get_todos():
    todos: Response = requests.get("...")
    todos.raise_for_status()

    return {"todos": todos.json()[:10]}


def lambda_handler(event: dict, context: LambdaContext) -> dict:
    return app.resolve(event, context)
```



```python
import requests
from requests import Response

from aws_lambda_powertools.event_handler import ALBResolver
from aws_lambda_powertools.utilities.typing import LambdaContext

app = ALBResolver()


@app.get("/todos")
@tracer.capture_method
def get_todos():
    todos: Response = requests.get("...")
    todos.raise_for_status()

    return {"todos": todos.json()[:10]}


def lambda_handler(event: dict, context: LambdaContext) -> dict:
    return app.resolve(event, context)
```

Idempotency

What is idempotency?

idem same
potent having power

idempotent *(ī-dem ˈpə-tən-sē)*
adjective

Same effect regardless of retries

Safely retry without side effects

Stable outcomes

Consistency

Resilience

Why do I need idempotency?

"The world is **asynchronous**."

Dr. Werner Vogels
VP and CTO at Amazon.com

Lambda retries

At-least-once delivery

Transient failures

SDK retries

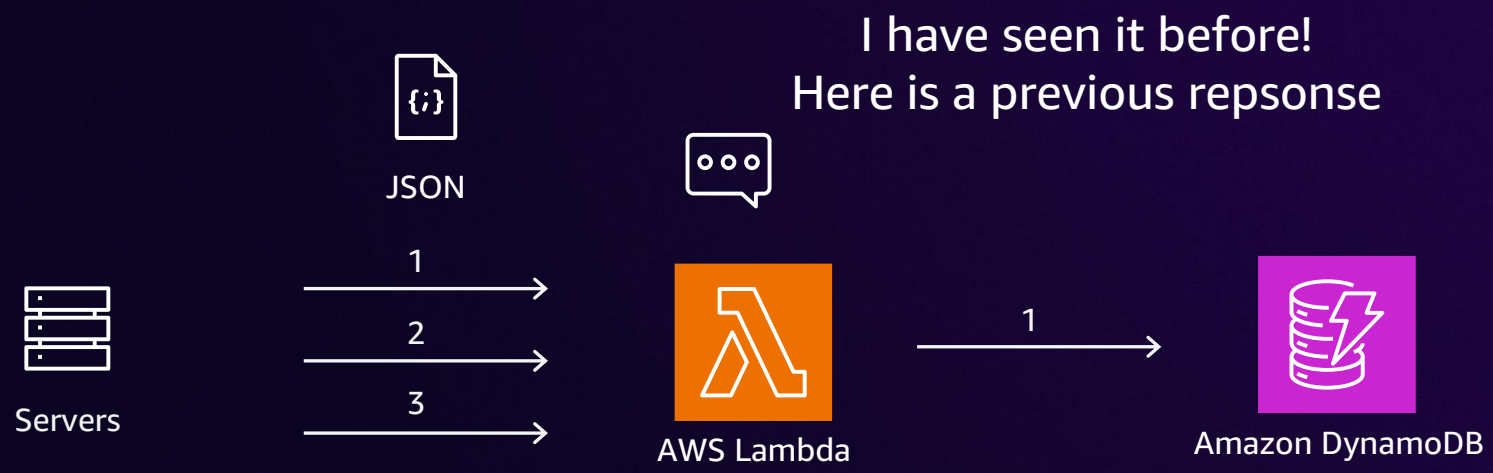
Eventual consistency

Auto-scaling operations

Idempotency

Idempotency

Idempotency



Timeout Concurrency Consistency Expiration

Validation Idempotency key Serialization

Scenario 1: First invocation

Caller

Your Lambda

Storage

Event

Get or set

key=id(event), status=INPROGRESS

Your code

set result={…}

key=id(event), status=COMPLETE

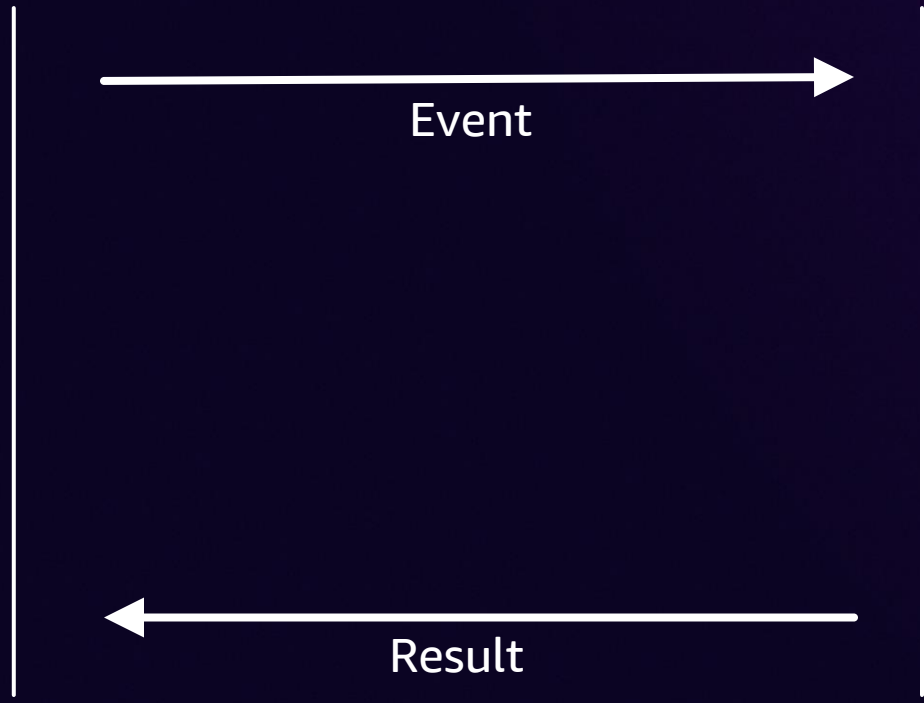
Result

Scenario 2: Second invocation

Caller

Your Lambda

Storage

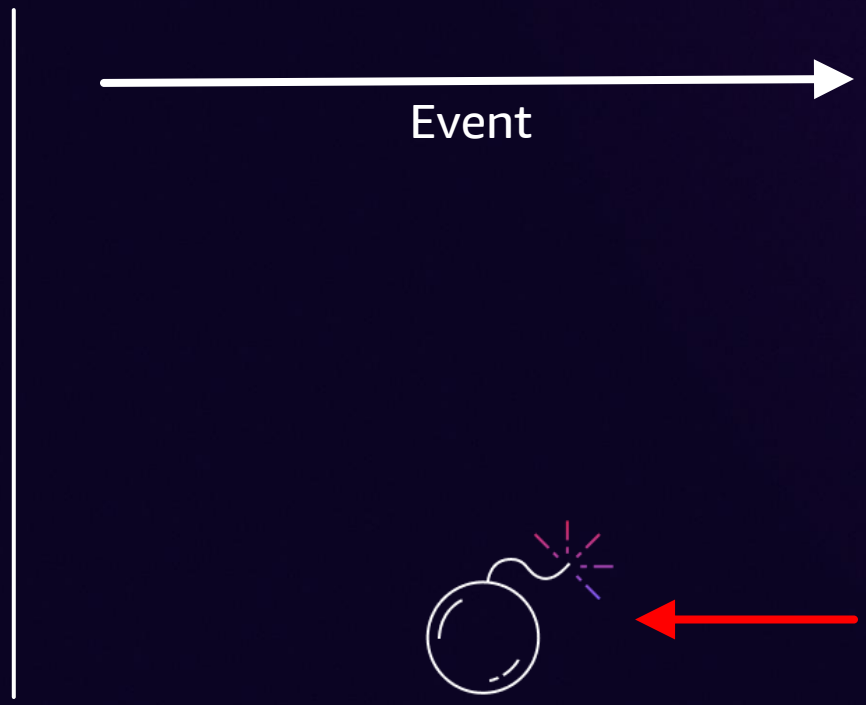


Scenario 3: Concurrent invocations

Caller

Your Lambda

Storage



Scenario 4: Handling exceptions

Caller

Your Lambda

Storage

Event

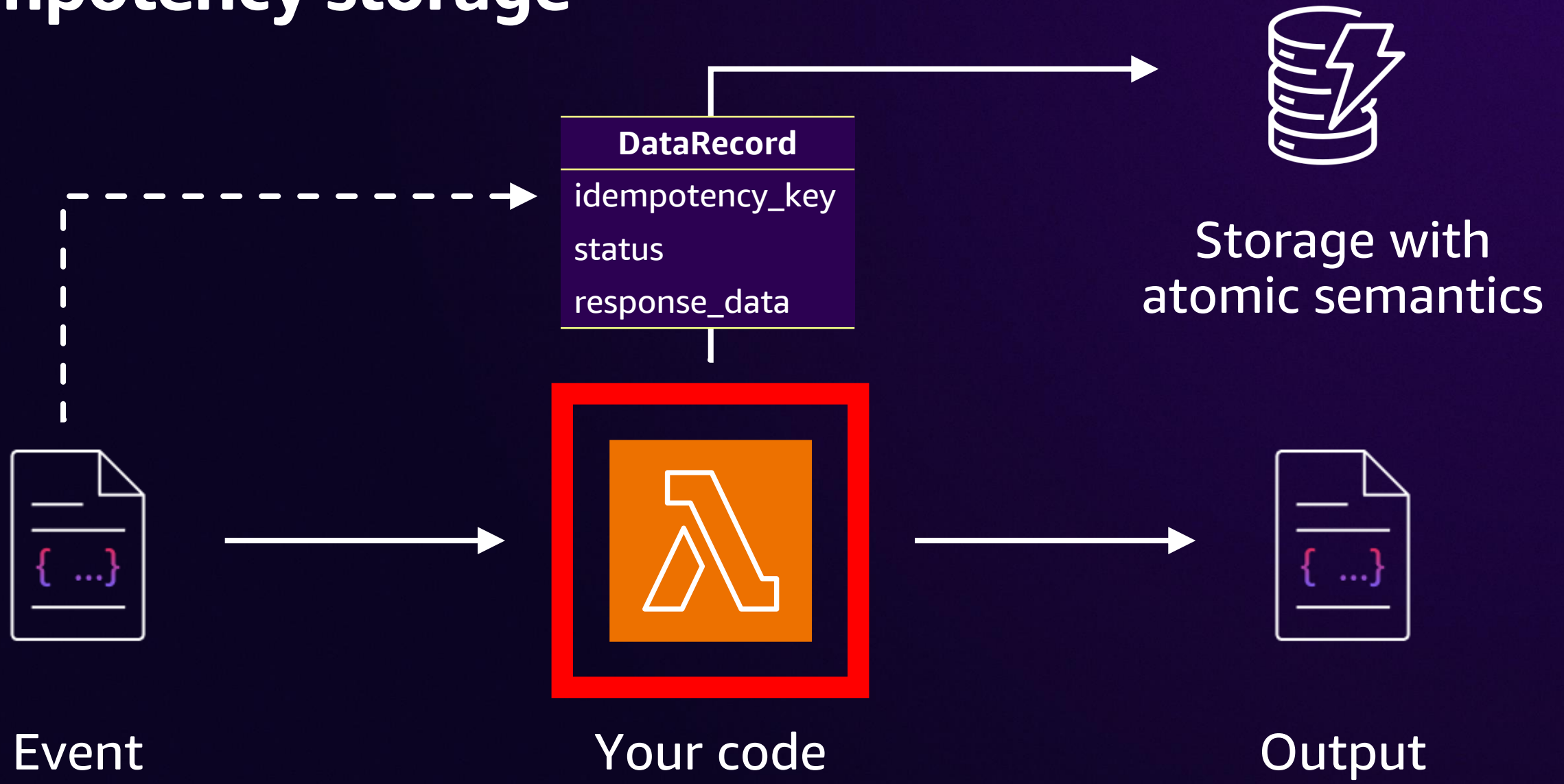
Get or set

key=id(event), status=INPROGRESS

Your code
CRASHES

delete key=id(event)

Idempotency storage



Idempotency

```python
from aws_lambda_powertools.utilities import LambdaContext
from aws_lambda_powertools.utilities.idempotency import (
    DynamoDBPersistenceLayer,
    idempotent,
)

ddbPersistenStore = DynamoDBPersistenceLayer(table_name="IdempotencyTable")

@idempotent(persistent_store=ddbPersistenStore)
def lambda_handler(event: dict, context: LambdaContext):
    try:
      payment = create_subscription_payment(event)
      return {
        "payment_id": payment.id,
        "statusCode": 200,
        "body": payment
      }
    except Exception as e:
      raise ValueError(f"Failed to create payment: {e}")

def create_subscription_payment(event: dict):
  # Create payment
  pass
```

Context awareness

```python
import os
from dataclasses import dataclass

from aws_lambda_powertools.utilities.idempotency import (
    DynamoDBPersistenceLayer,
    IdempotencyConfig,
    idempotent_function,
)
from aws_lambda_powertools.utilities.typing import LambdaContext

table = os.getenv("IDEMPOTENCY_TABLE", "")
dynamodb = DynamoDBPersistenceLayer(table_name=table)
config = IdempotencyConfig(event_key_jmespath="order_id")


@idempotent_function(data_keyword_argument="order", config=config, persistence_store=dynamodb)
def process_order(order: Order):
    return f"processed order {order.order_id}"


def lambda_handler(event: dict, context: LambdaContext):
    config.register_lambda_context(context)

    order_item = OrderItem(sku="fake", description="sample")
    order = Order(item=order_item, order_id=1)

    process_order(order=order)
```

Fine-tune settings

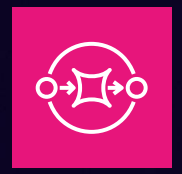
```python
import os

from aws_lambda_powertools.utilities.idempotency import (
    DynamoDBPersistenceLayer,
    IdempotencyConfig,
    idempotent,
)
from aws_lambda_powertools.utilities.typing import LambdaContext

table = os.getenv("IDEMPOTENCY_TABLE", "")
persistence_layer = DynamoDBPersistenceLayer(table_name=table)
config = IdempotencyConfig(
    event_key_jmespath="powertools_json(body)",
    # by default, it holds 256 items in a Least-Recently-Used (LRU) manner
    use_local_cache=True,
)


@idempotent(config=config, persistence_store=persistence_layer)
def lambda_handler(event, context: LambdaContext):
    return event
```

Swap persistence layer

```python
import os

from aws_lambda_powertools.utilities.idempotency import (
    idempotent,
)
from aws_lambda_powertools.utilities.typing import LambdaContext
from aws_lambda_powertools.utilities.idempotency.persistence.redis import (
    RedisCachePersistenceLayer,
)

valkey_endpoint = os.getenv("VALKEY_ENDPOINT")
persistence_layer = RedisCachePersistenceLayer(host=valkey_endpoint, port=6379)


@idempotent(persistence_store=persistence_layer)
def lambda_handler(event, context: LambdaContext):
  return event
```

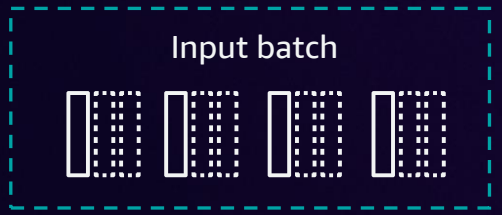
Batch processing

Batch processing



Amazon Simple Queue Service (Amazon SQS)

Amazon Kinesis

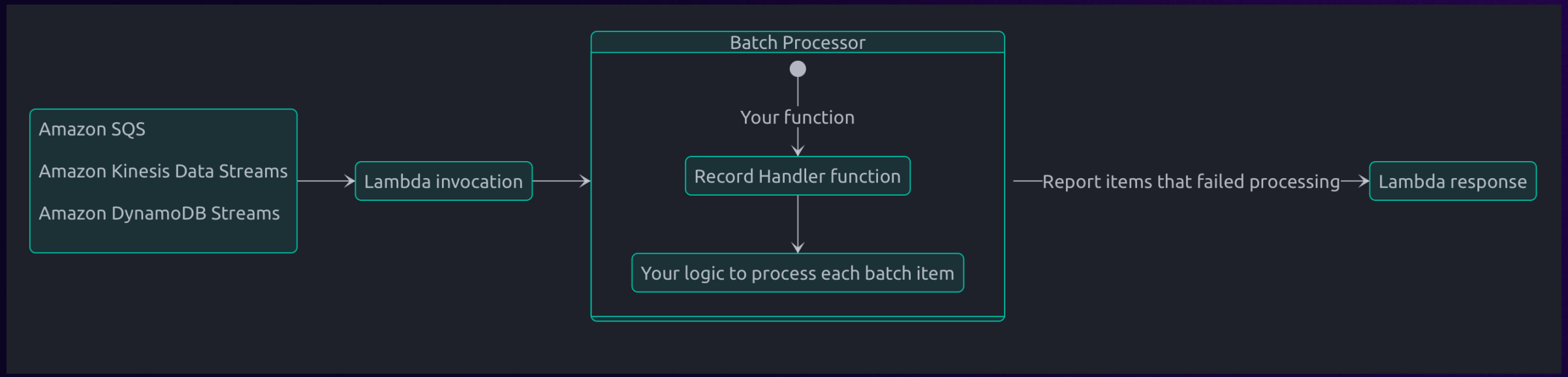


AWS Lambda

Amazon DynamoDB

```
{
  "batchItemFailures": [
    {
      "itemIdentifier": "244fc6b4-87a3-44ab-83d2-361172410c3a"
    }
  ]
}
```

Batch processing



Batch processing

```python
from aws_lambda_powertools import Logger, Tracer
from aws_lambda_powertools.utilities.batch import (
    BatchProcessor,
    EventType,
    process_partial_response,
)
from aws_lambda_powertools.utilities.data_classes.sqs_event import SQSRecord
from aws_lambda_powertools.utilities.typing import LambdaContext

processor = BatchProcessor(event_type=EventType.SQS)
tracer = Tracer()
logger = Logger()


@tracer.capture_method
def record_handler(record: SQSRecord):
    payload: str = record.json_body  # if json string data, otherwise record.body for str
    logger.info(payload)


@logger.inject_lambda_context
@tracer.capture_lambda_handler
def lambda_handler(event, context: LambdaContext):
    return process_partial_response(
        event=event,
        record_handler=record_handler,
        processor=processor,
        context=context,
    )
```

Batch processing

```python
from aws_lambda_powertools import Logger, Tracer
from aws_lambda_powertools.utilities.batch import (
  BatchProcessor,
  EventType,
  process_partial_response,
)
from aws_lambda_powertools.utilities.data_classes.kinesis_stream_event import (
    KinesisStreamRecord,
)
from aws_lambda_powertools.utilities.typing import LambdaContext

processor = BatchProcessor(event_type=EventType.KinesisDataStreams)
tracer = Tracer()
logger = Logger()


@tracer.capture_method
def record_handler(record: KinesisStreamRecord):
  logger.info(record.kinesis.data_as_text)
  payload: dict = record.kinesis.data_as_json()
  logger.info(payload)

@logger.inject_lambda_context
@tracer.capture_lambda_handler
def lambda_handler(event, context: LambdaContext):
  return process_partial_response(
    event=event,
    record_handler=record_handler,
    processor=processor,
    context=context
  )
```

Roadmap

Roadmap

Powertools for AWS Lambda (Python)

Powertools for AWS Lambda (TypeScript)

Powertools for AWS Lambda (.NET)

Thank you!

Please complete the session survey in the mobile app

Andrea Amorosi

@dreamorosi