AWS serverless with Lambda, CDK, and DynamoDB

You build the workshop shop order flow serverless on AWS, with Lambda, DynamoDB, events, and Step Functions, all as code in CDK.

Mike Bild

Exercises: github.com/MikeBild/introduction-aws

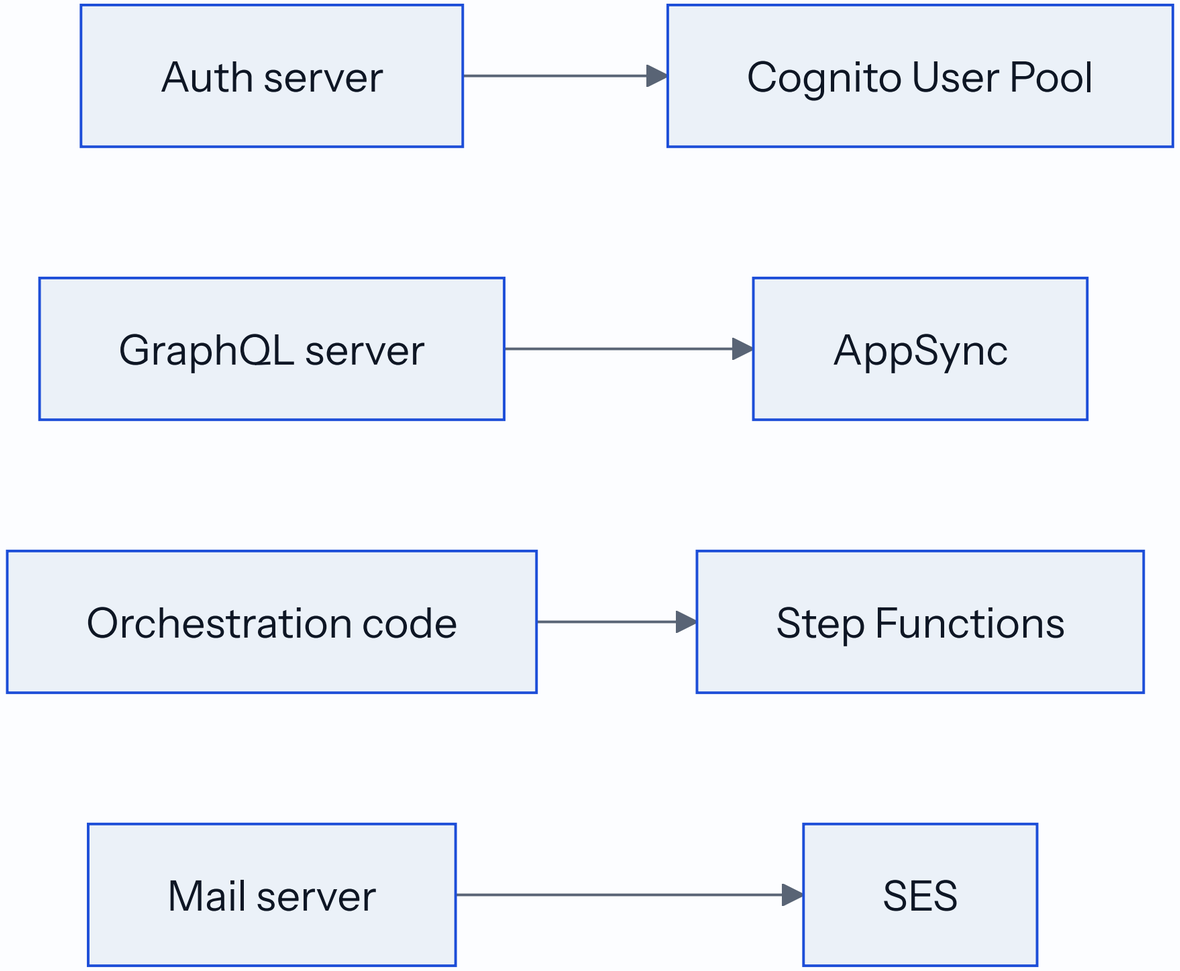
Mindset, Lambda, CDK, IAM

What does serverless really take off your plate? · A Lambda function is just a handler · Infrastructure belongs as TypeScript in the repo · Each function gets only its own permissions

Describe, don’t operate

- Managed Services instead of your own servers
- Configuration instead of infrastructure maintenance
- Billing by usage instead of provisioning

The recurring pattern



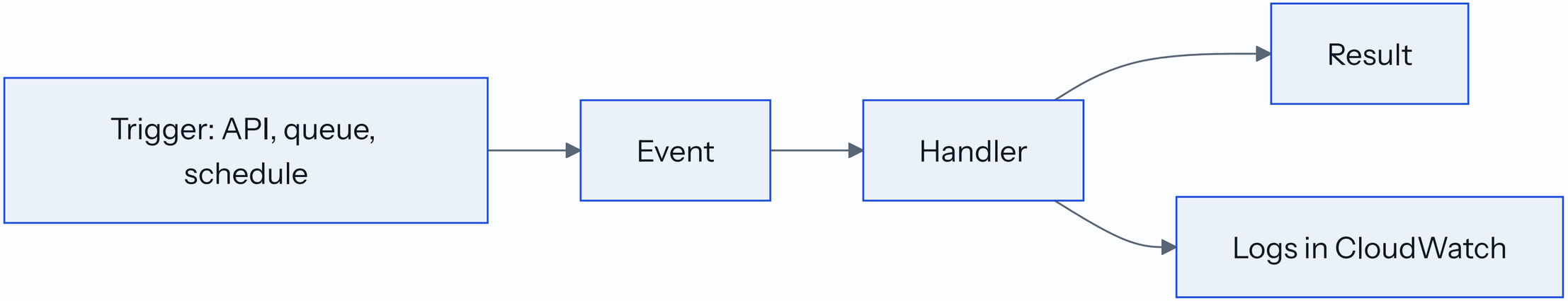
Pay per request

- Lambda bills requests and runtime multiplied by memory
- DynamoDB On-Demand bills reads and writes
- Without traffic, the application costs almost nothing

Where does serverless hit limits?

- Coordinating many functions quickly becomes confusing
- Cold Starts are part of the execution model
- You model data by Access Patterns, not by normal form

Event in, result out



Init phase and invoke phase

- Init runs once per execution environment
- Invoke runs for every request
- Expensive objects belong in the init phase

Limits you need to know

- At most fifteen minutes of runtime per invocation
- At most six megabytes of payload for a synchronous invocation
- No state between invocations that you can rely on

A handler in TypeScript

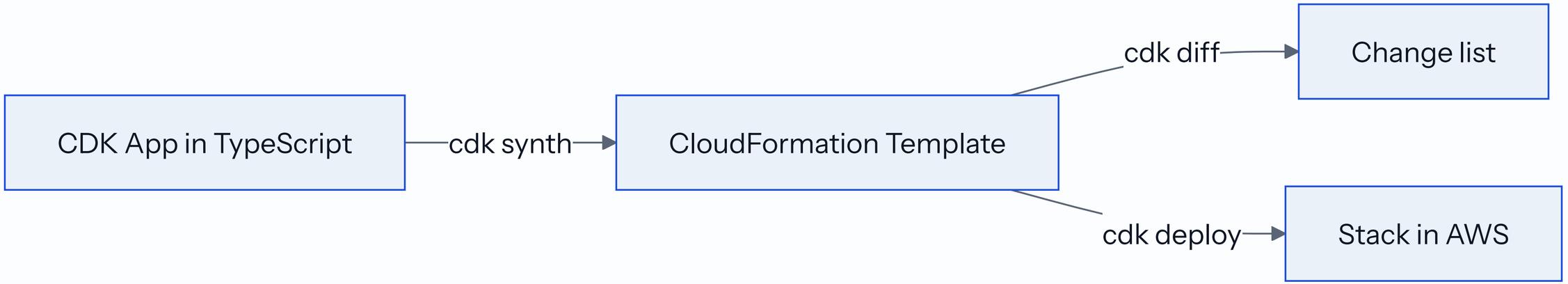
```ts
import type { APIGatewayProxyHandlerV2 } from 'aws-lambda'
import { parseOrder, totalCents } from './order.ts'

export const handler: APIGatewayProxyHandlerV2 = async (event) => {
  const order = parseOrder(event.body)
  if (!order)
    return { statusCode: 400, body: JSON.stringify({ message: 'invalid order' }) }
  return { statusCode: 200, body: JSON.stringify({ totalCents: totalCents(order) }) }
}
```

Why not click it?

- Clicks cannot be reviewed
- Clicks cannot be repeated
- Clicks cannot be rolled back

From TypeScript to CloudFormation



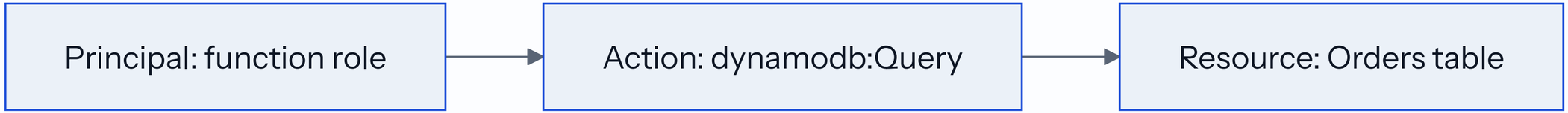
Constructs at three levels

- L1: CloudFormation one to one, Cfn prefix
- L2: useful defaults and grant methods
- L3: ready-made patterns from multiple resources

A stack in CDK v2

```
export class OrdersStack extends Stack {
  constructor(scope: Construct, id: string) {
    super(scope, id)
    const table = new TableV2(this, 'Orders', {
      partitionKey: { name: 'pk', type: AttributeType.STRING },
    })
    const placeOrder = new NodejsFunction(this, 'PlaceOrder', {
      entry: 'src/place-order.ts',
      runtime: Runtime.NODEJS_24_X,
      architecture: Architecture.ARM_64,
      environment: { TABLE_NAME: table.tableName },
    })
    table.grantReadWriteData(placeOrder)
  }
}
```

Who may do what to what?



One role per function

- Each function automatically gets its own role
- Shared roles collect all permissions over time
- With its own role, a leak affects only one function

grant instead of wildcard

```
// zu breit
listOrders.addToRolePolicy(new PolicyStatement({
  actions: ['dynamodb:*'],
  resources: ['*'],
}))

// genau passend
table.grantReadData(listOrders)
```

HTTP API, DynamoDB, configuration

How does an HTTP request reach a function? · Access patterns first, then the table · When is single-table design worth it? · Where should configuration and secrets go?

Three ways to the function

- HTTP API: lean, low cost, for most web apps
- REST API: more features, higher cost
- Function URL: one endpoint per function, without routing

How many functions per API?

- One function per route: small permissions, many functions
- One function per resource: a good middle ground
- One function for everything: fewer cold starts, broad permissions

Routes in CDK

```
const api = new HttpApi(this, 'OrdersApi', {
  corsPreflight: {
    allowOrigins: ['https://app.example.com'],
    allowMethods: [CorsHttpMethod.GET, CorsHttpMethod.POST],
  },
})
api.addRoutes({
  path: '/orders',
  methods: [HttpMethod.POST],
  integration: new HttpLambdaIntegration('PlaceOrder', placeOrder),
})
```

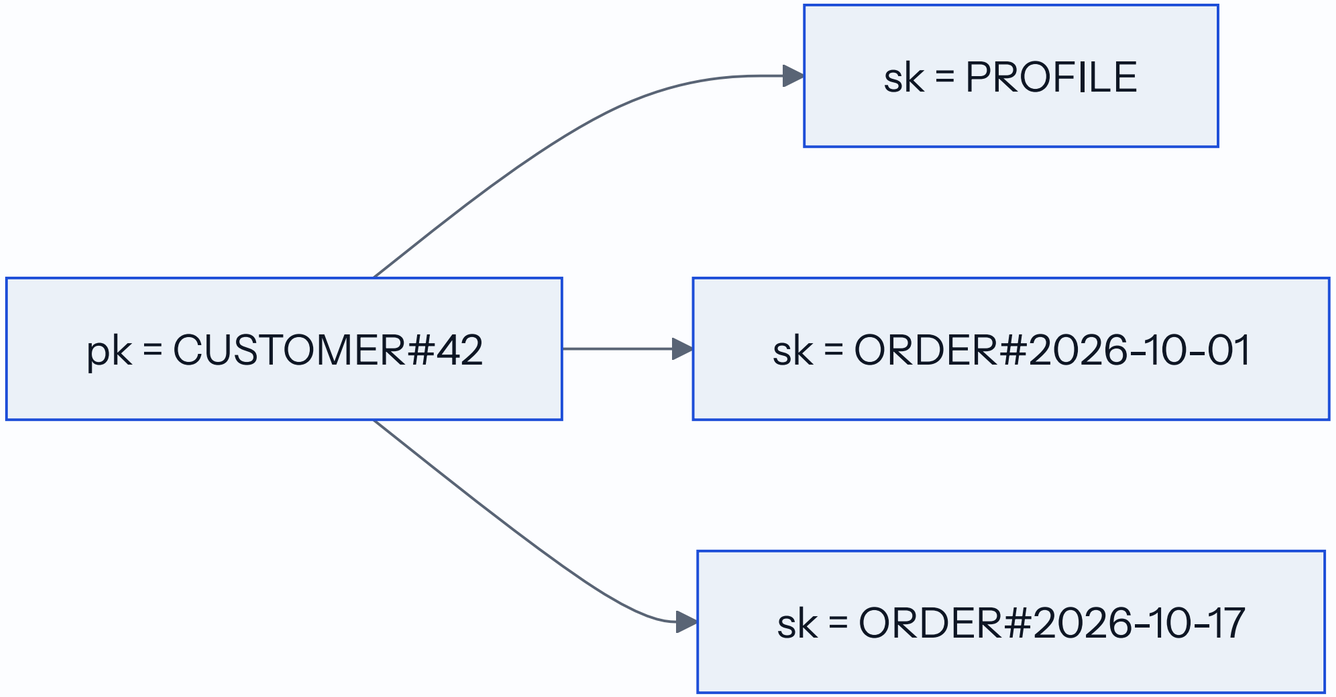
The event in the handler

```
export const handler: APIGatewayProxyHandlerV2 = async (event) => {
  const orderId = event.pathParameters?.orderId
  const limit = Number(event.queryStringParameters?.limit ?? 20)
  const order = await findOrder(orderId!, limit)
  return order
    ? { statusCode: 200, body: JSON.stringify(order) }
    : { statusCode: 404, body: JSON.stringify({ message: 'not found' }) }
}
```

The reversed order

- SQL: normalize first, query freely later
- DynamoDB: collect questions first, then build keys
- New questions later cost new indexes or migrations

Partition key and sort key



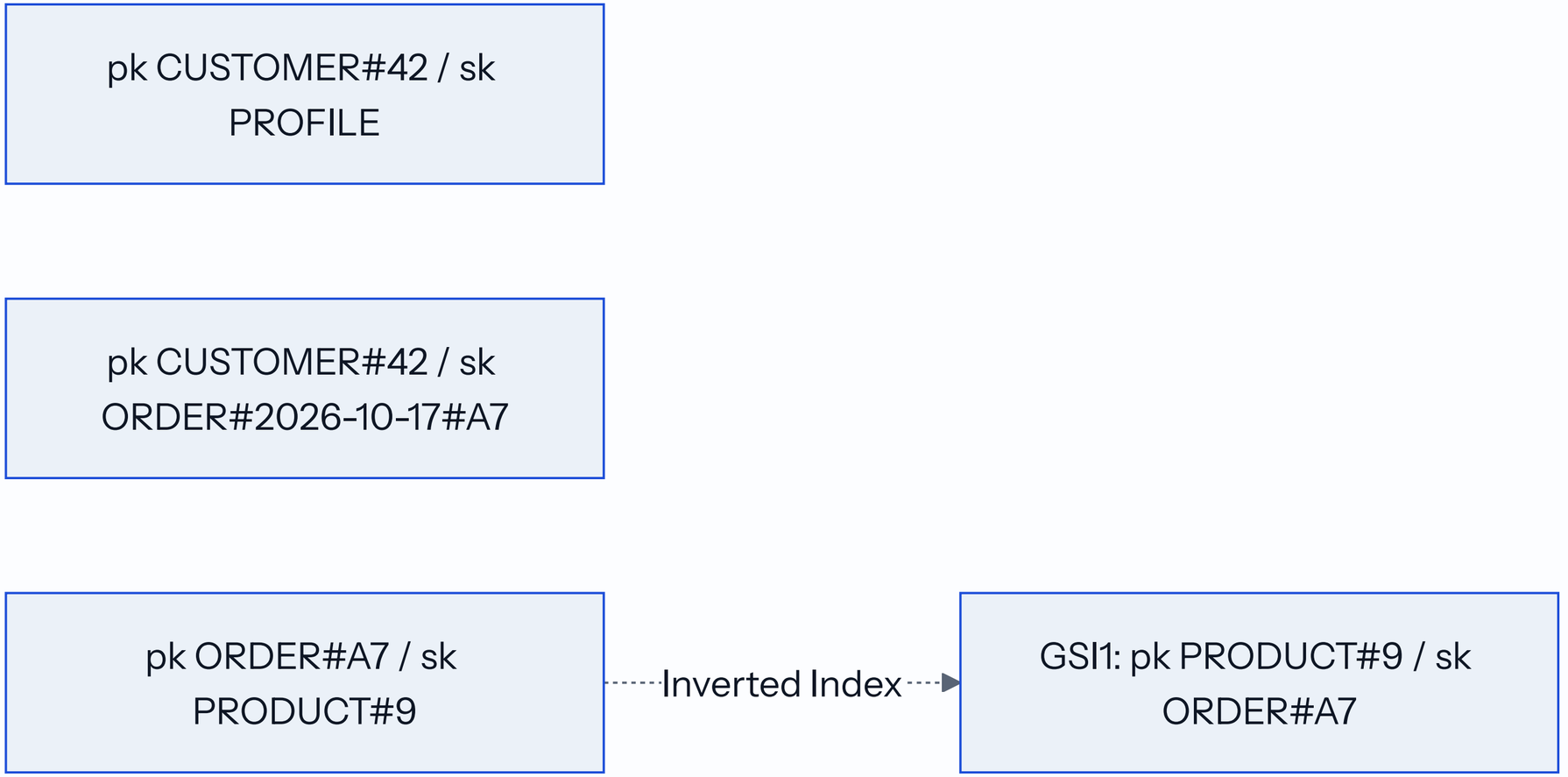
New questions with a GSI

- GSI: its own partition key and sort key, addable anytime
- LSI: same partition key, only when creating the table
- Both cost additional writes

Query instead of scan

```
const { Items } = await db.send(new QueryCommand({
  TableName: tableName,
  KeyConditionExpression: 'pk = :pk AND begins_with(sk, :order)',
  ExpressionAttributeValues: {
    ':pk': `CUSTOMER#${customerId}`,
    ':order': 'ORDER#',
  },
  ScanIndexForward: false,
  Limit: 20,
}))
```

One layout for multiple entities



Why there is no join

- Joins do not scale with predictable latency
- Related data belongs in an item collection
- Duplicates are intentional, not accidental

When is it overdone?

- Access patterns are still unclear or change often
- Analytics and ad hoc questions need SQL
- GraphQL resolvers resolve fields individually anyway

The FilterExpression trap

```
// reads all orders for the customer, filters only afterward
new QueryCommand({
  TableName: tableName,
  KeyConditionExpression: 'pk = :pk',
  FilterExpression: '#status = :placed',
  ExpressionAttributeNames: { '#status': 'status' },
  ExpressionAttributeValues: { ':pk': 'CUSTOMER#42', ':placed': 'placed' },
})
```

Configuration is not a secret

- Configuration: table name, region, feature flag
- Secret: API key, password, signing key
- Configuration as an environment variable, secrets from a store

Two stores to choose from

- Parameter store: low cost, for configuration and simple secrets
- Secrets Manager: with rotation, for database credentials
- Both with IAM permissions per parameter

Load with a cache

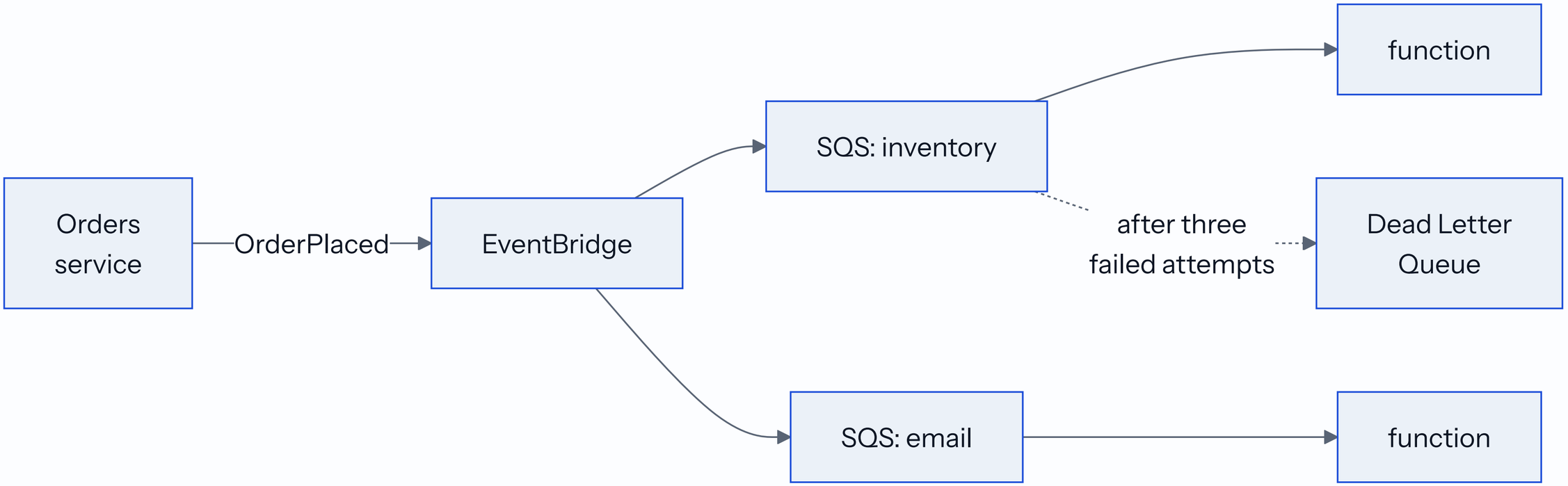
```
import { getSecret } from '@aws-lambda-powertools/parameters/secrets'
import { getParameter } from '@aws-lambda-powertools/parameters/ssm'

export const handler = async (event: PaymentEvent) => {
  const apiKey = await getSecret('payments/api-key', { maxAge: 300 })
  const limit = await getParameter('/orders/max-items', { maxAge: 300 })
  return chargePayment({ apiKey: apiKey!, limit: Number(limit), event })
}
```

Events, email, workflows, cold starts

Every message can arrive twice · With SES, the effort is in DNS, not code · Workflows as state machines, not call chains · Measure cold starts instead of fearing them

Queue or event?



At least once means idempotent

- SQS and EventBridge deliver at least once
- Duplicate messages are normal, not an error
- An idempotency key prevents duplicate effects

Publish an event

```
import { EventBridgeClient, PutEventsCommand } from '@aws-sdk/client-eventbridge'

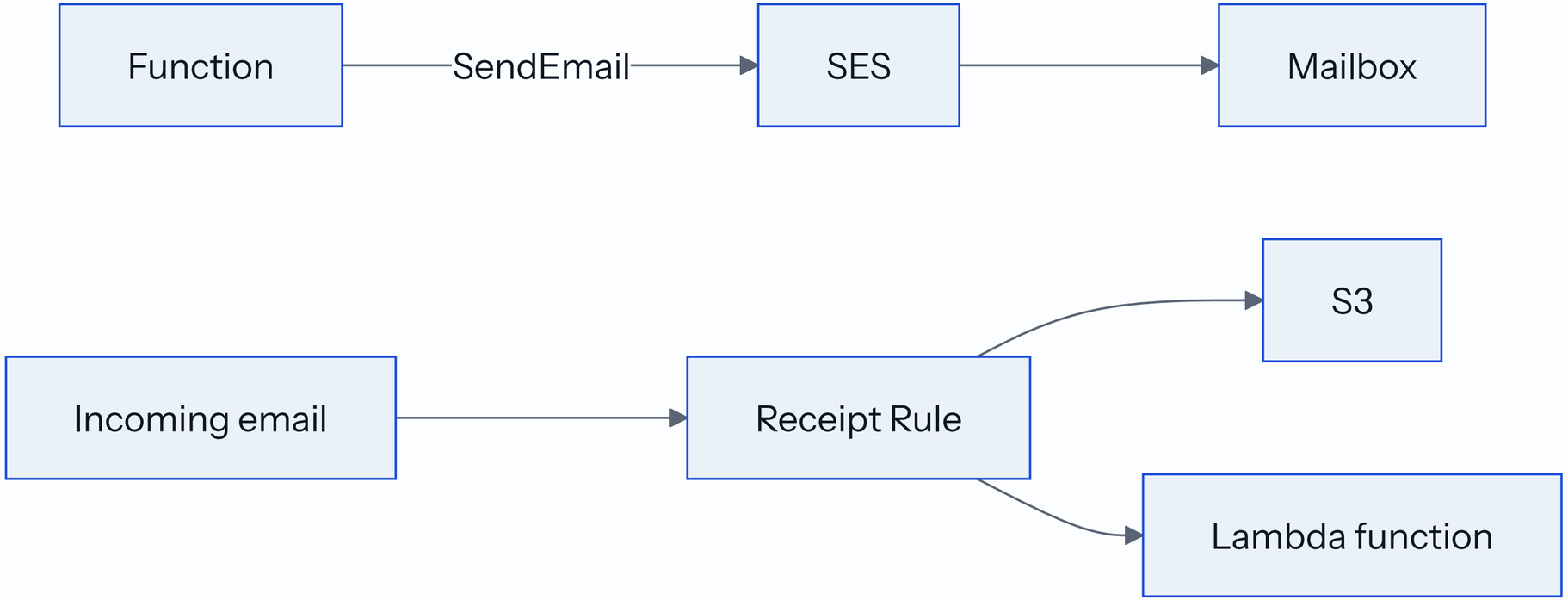
const events = new EventBridgeClient({})

export const publishOrderPlaced = (order: Order) =>
  events.send(new PutEventsCommand({
    Entries: [{
      EventBusName: process.env.EVENT_BUS_NAME,
      Source: 'orders',
      DetailType: 'OrderPlaced',
      Detail: JSON.stringify({ orderId: order.id, customerId: order.customerId }),
    }],
  }))
```

Batches with partial failures

```
export const handler: SQSHandler = async (event) => {
  const batchItemFailures: SQSBatchItemFailure[] = []
  for (const record of event.Records) {
    try {
      await reserveStock(JSON.parse(record.body))
    } catch {
      batchItemFailures.push({ itemIdentifier: record.messageId })
    }
  }
  return { batchItemFailures }
}
```

Two directions



Three organizational hurdles

- Verify the domain with DKIM in DNS
- Leave the sandbox by requesting access from AWS
- Evaluate bounces and complaints

Process incoming email

- The event contains only headers and metadata
- The raw email is in S3 under the message ID
- The S3 action must come before the Lambda action

Send a confirmation

```
import { SESv2Client, SendEmailCommand } from '@aws-sdk/client-sesv2'

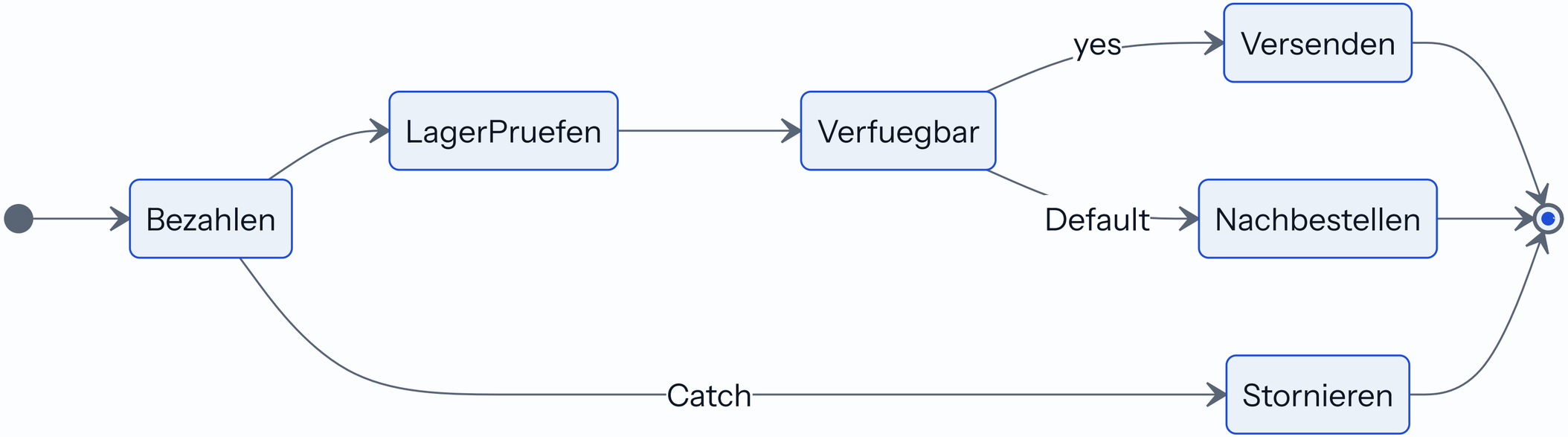
const ses = new SESv2Client({})

export const sendConfirmation = (order: Order, customer: Customer) =>
  ses.send(new SendEmailCommand({
    FromEmailAddress: 'shop@example.com',
    Destination: { ToAddresses: [customer.email] },
    Content: {
      Template: {
        TemplateName: 'order-confirmed',
        TemplateData: JSON.stringify({ orderId: order.id }),
      },
    },
  }))
```

Where is your workflow today?

- Function A calls B, B calls C, C calls A back
- Retry and error paths are hidden in each handler
- Nobody sees where an order is currently stuck

The workflow as a state machine



Standard or Express?

- Standard: up to one year, exactly once, full history
- Express: up to five minutes, very high rates, logs in CloudWatch
- The type is fixed when you create it

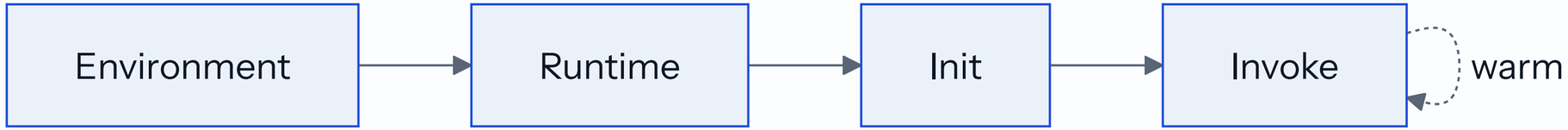
When not to use it?

- A single step without an error path
- Pure data flows without decisions
- Very high rates with a budget of only a few milliseconds

A Task with Retry and Catch

```
const charge = LambdaInvoke.jsonata(this, 'Charge', {
  lambdaFunction: chargePayment,
  payload: TaskInput.fromObject({ orderId: '{% $states.input.orderId %}' }),
  outputs: '{% $states.result.Payload %}',
})
charge.addRetry({
  errors: ['PaymentProviderTimeout'],
  interval: Duration.seconds(2),
  backoffRate: 2,
  maxAttempts: 3,
})
charge.addCatch(cancelOrder, { errors: ['PaymentDeclined'] })
```

What a cold start is



First shrink the bundle and init code

- Keep the bundle small with esbuild and targeted imports
- Create clients once at module level
- Load rarely used code only when needed

More memory also brings more CPU

- Lambda has only one control: memory
- Too little memory makes CPU-heavy code slow
- For I/O-heavy load, more memory stops helping

Pre-warmed with Provisioned Concurrency

- Keeps environments ready and initialized
- Costs money even without traffic
- Pays off only for predictable, latency-critical load

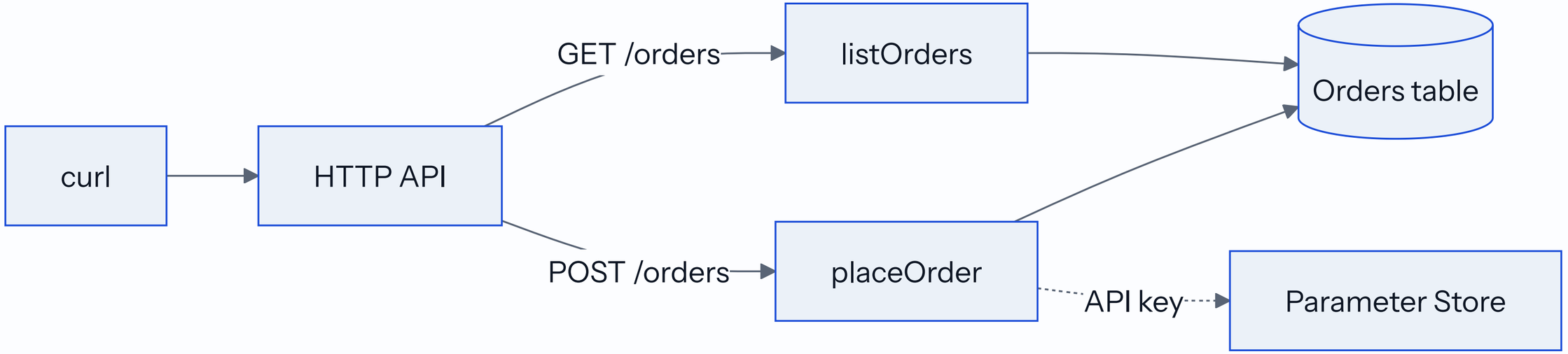
Measure in CloudWatch

```
REPORT RequestId: 3f1c…  Duration: 41.20 ms  Billed Duration: 354 ms
Memory Size: 1024 MB  Max Memory Used: 98 MB  Init Duration: 312.45 ms
```

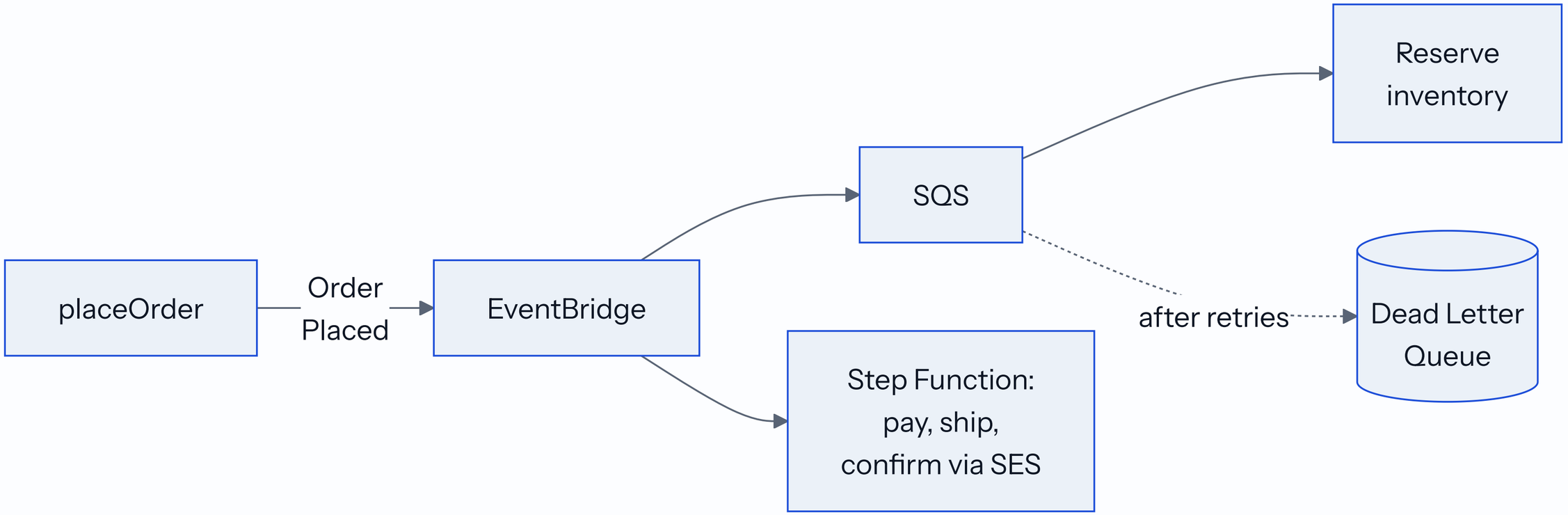
Practice: deploy the workshop shop end to end

Hands-on part one, the order reaches the table · Hands-on part two, from event to shipping

One stack for the order



After the request, work continues



Questions?

Mike Bild · mike.bild@gmail.com · mikebild.dev

Workshops für Teams, die Architekturentscheidungen selbst treffen