Serverless NFRs

Robert Bulmer
Software Architect @ City Electric

Electrical Wholesaler (🇬🇧🇮🇪🇺🇸🇨🇦🇪🇸🇦🇺🇳🇱)

What are NFRs?

Functional Requirements

Functional requirements are the customer / business requirements.

i.e. System must provide a CSV file of all active customers within the user specified date range

What are NFRs?

Non-Functional Requirements

Non-Functional requirements are the technical system requirements that need to be met to fulfill those functional requirements.

i.e. System must provide a CSV file of all active customers within the user specified date range -> ?

The system must be able to allow 10 concurrent users to generate the file.

The system must be able to produce a file for our growing customer base of 1,500,000 customers.

NFR Categories

Performance	Scalability	Usability
Reliability	Maintainability	Portability
Interoperability	...Security, Compliance, Cost, Monitoring and Auditing.	

A.K.A – The ‘ilities’

My Key NFRs

- ✓ Maintainability
- ✓ Portability
- ✓ Cost
- ✓ **Performance**
- ✓ **Scalability**

Maintainability

Often, we want to make sure a system can be maintained and used again.

Maintainability consists of requirements to do with ***reusability*** and ***analysability.***

Reusability, can part of the system be used by another system.

With **Analysability**, can we easily identify faults and improvements we need to make.

Maintainability

Example (*Reusability*):

Functional Requirement: The integration must convert the Country ISO Code to the ISO Name – i.e. GBR to United Kingdom of Great Britain and Northern Ireland

Non-Functional Requirement: All standard data transformations are reusable across services.

***Implementation**: Publish a shared utility library to the company registry (AWS Code Artifact) to transform standard ISO values*

Maintainability

Example (*Reusability*):

Functional Requirement: The integration must transform Customer data into the format required by the downstream integration.

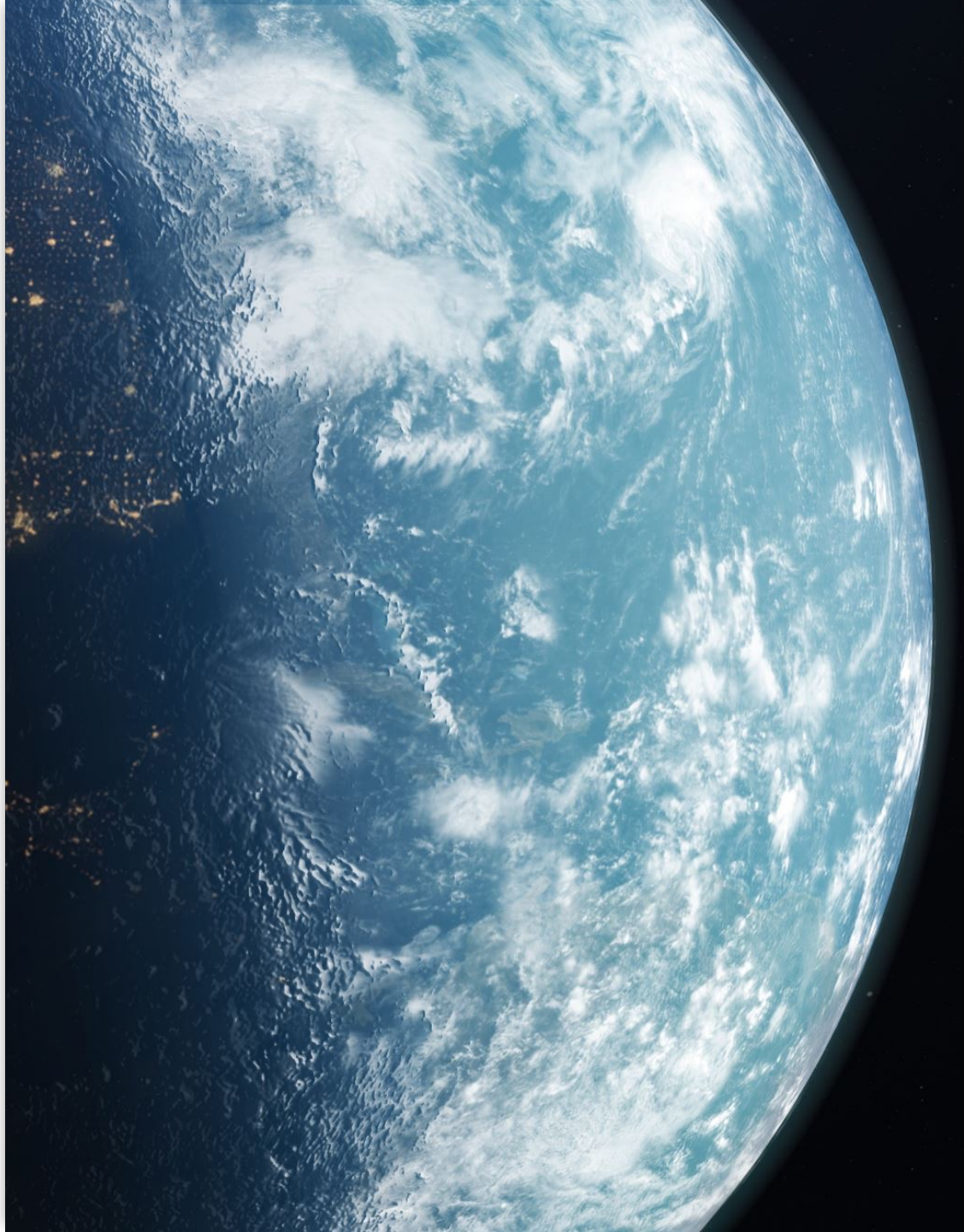
Non-Functional Requirement: All architecture patterns that follow the company standards must be reusable across services.

***Implementation**: The integration should create an AWS CDK Construct to allow reuse of the IAC.*

Portability

With portability we often ask ourselves;

- ***Adaptability*** can we adapt this for another environment?
- ***Installability*** can we deploy and install the system on another platform or in another region or account.



Portability

Serverless Internationalisation 🚀

Example of building our serverless solutions once and with extensibility so they can be consumed in different locales safely, written in the AWS CDK and Typescript.

Serverless Advocate · Follow

11 min read · Jun 30, 2022

Example (*Adaptability*):

Functional Requirement: The integration must support UKI and NA customers

Non-Functional Requirement: *Use Internationalisation in the code base to adapt the deployment to the correct region*

Portability

Example (*Installability*):

Functional Requirement: The integration must support UKI and NA customers

Non-Functional Requirement: *The AWS CDK app must be configurable to deploy separately to multiple AWS regions*

Cost

No business has infinite budget. During normal operation it is not uncommon to be asked how much your application is going to cost the company.

They often want a monthly, yearly and future growth figure.

Cost

Example (*Cost*):

Functional Requirement: The integration must support our 100,000 Customer base

Non-Functional Requirement: *The integration must cost less than $1,000/month (est. using the AWS pricing calculator).*

Implementation: Must have AWS budget alerts enabled and the application must be tagged for suitable identification

Performance

With performance we often look at ***response time*** and ***throughput.***

Response time being how quickly the application reacts to input or an event.

Throughput is typically how many transactions can we process within a given time frame.

Performance

When should we use Serverless?

When you have periodic unpredictable, spiky, workloads. *There are other use cases where you might want to leverage the agility and developer experience that Serverless provides.*

If you're comparing serverless instances to regular provisioned instances, then you're comparing apples and oranges… They are for two different use cases. **Think "Serverless-first" not "serverless-only"**

Typically, when people say performance, they want to know exact milliseconds of the database queries. While that contributes to the performance NFR… we often need to look at a higher level first.

Performance – Response Time

What response time do we need from our system?

Typically, web application ***users expect a Flow/Continuity response within 1 second***; This usually involves refreshing data, reloading web pages etc.

Artillery
cloud-scale load testing

https://www.npmjs.com/package/artillery

artillery > ! performance-test.yml

```yaml
config:
  target: '{{$env.API_URL}}'
  phases:
    - duration: 10 # Length of test
      arrivalRate: 1 # Number of users
      rampTo: 1 # Increase Users over time
      name: Test Phase
  plugins:
    ensure: {}
    apdex: {}
    metrics-by-endpoint: {}
  apdex:
    threshold: 500
  ensure:
    thresholds:
      - http.response_time.p99: 500
      - http.response_time.p95: 300
scenarios:
  - flow:
      - get:
          url: '/customer/{{$env.ID_UNDER_TEST}}'

```

Artillery
cloud-scale load testing

https://www.npmjs.com/package/artillery

Apdex- Application Performance Index

```
--------------------------------
Summary report @ 10:19:45(+0100)
--------------------------------

apdex.frustrated: ..... 4
apdex.satisfied: ..... 6
apdex.tolerated: ..... 0
http.codes.200: ..... 10
http.downloaded_bytes: ..... 1130
http.request_rate: ..... 1/sec
http.requests: ..... 10
http.response_time:
  min: ..... 61
  max: ..... 3334
  mean: ..... 1310.2
  median: ..... 89.1
  p95: ..... 3197.8
  p99: ..... 3197.8
http.responses: ..... 10
plugins.metrics-by-endpoint./prod//customer/403fdfbd-d1c2-405a-a6e5-9540775b... 10
plugins.metrics-by-endpoint.response_time./prod//customer/403fdfbd-d1c2-405a-a6e5-9540775baba3:
  min: ..... 61
  max: ..... 3334
  mean: ..... 1310.2
  median: ..... 89.1
  p95: ..... 3197.8
  p99: ..... 3197.8
vusers.completed: ..... 10
vusers.created: ..... 10
vusers.created_by_name.0: ..... 10
vusers.failed: ..... 0
vusers.session_length:
  min: ..... 139.4
  max: ..... 3489.4
  mean: ..... 1388.2
  median: ..... 159.2
  p95: ..... 3262.4
  p99: ..... 3262.4

Checks:
fail: http.response_time.p95 < 300
fail: http.response_time.p99 < 500
```

Artillery
cloud-scale load testing

https://www.npmjs.com/package/artillery

artillery > ! performance-test.yml

```yaml
config:
  target: '{{$env.API_URL}}'
  phases:
    - duration: 1 # Length of test
      arrivalRate: 1 # Number of users
      rampTo: 1 # Increase Users over time
      name: WarmUp Phase
    - pause: 5
      name: WarmUp Wait Phase
    - duration: 10 # Length of test
      arrivalRate: 1 # Number of users
      rampTo: 1 # Increase Users over time
      name: Test Phase
  plugins:
    ensure: {}
    apdex: {}
    metrics-by-endpoint: {}
  apdex:
    threshold: 500
  ensure:
    thresholds:
      - http.response_time.p99: 500
      - http.response_time.p95: 300
scenarios:
  - flow:
      - get:
          url: '/customer/{{$env.ID_UNDER_TEST}}'

```

1 failed... = Lambda Cold Starts

Artillery
cloud-scale load testing

https://www.npmjs.com/package/artillery

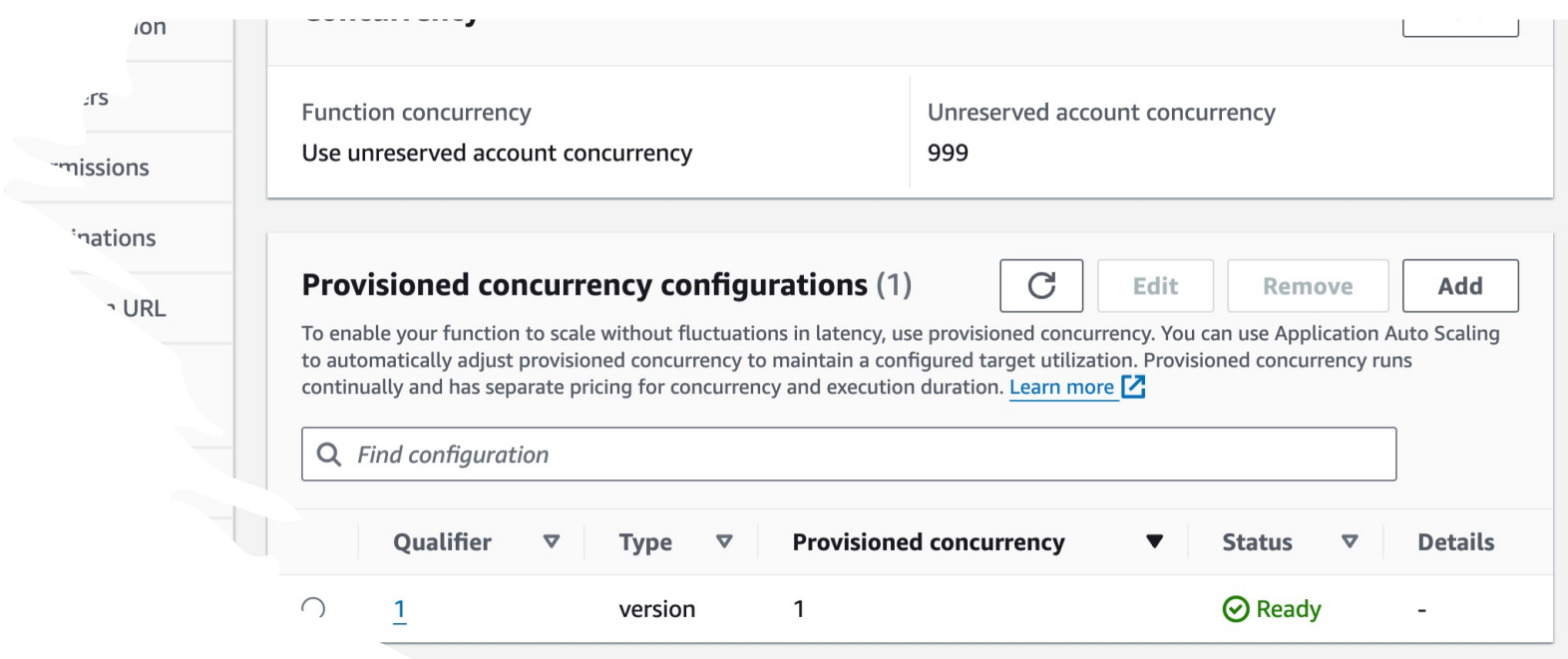
```
--------------------------------
Summary report @ 11:07:07(+0100)
--------------------------------

apdex.frustrated: ..... 1
apdex.satisfied: ..... 10
apdex.tolerated: ..... 0
http.codes.200: ..... 11
http.downloaded_bytes: ..... 1243
http.request_rate: ..... 1/sec
http.requests: ..... 11
http.response_time:
  min: ..... 52
  max: ..... 3178
  mean: ..... 369
  median: ..... 85.6
  p95: ..... 162.4
  p99: ..... 162.4
http.responses: ..... 11
plugins.metrics-by-endpoint./prod//customer/403fdfbd-d1c2-405a-a6e5-9540775b... 11
plugins.metrics-by-endpoint.response_time./prod//customer/403fdfbd-d1c2-405a-a6e5-9540775baba3:
  min: ..... 52
  max: ..... 3178
  mean: ..... 369
  median: ..... 85.6
  p95: ..... 162.4
  p99: ..... 162.4
vusers.completed: ..... 11
vusers.created: ..... 11
vusers.created_by_name.0: ..... 11
vusers.failed: ..... 0
vusers.session_length:
  min: ..... 116.5
  max: ..... 3306.9
  mean: ..... 448.5
  median: ..... 156
  p95: ..... 232.8
  p99: ..... 232.8

Checks:
ok: http.response_time.p95 < 300
ok: http.response_time.p99 < 500

Apdex score: 0.9090909090909091 (good)
```

Cold Starts

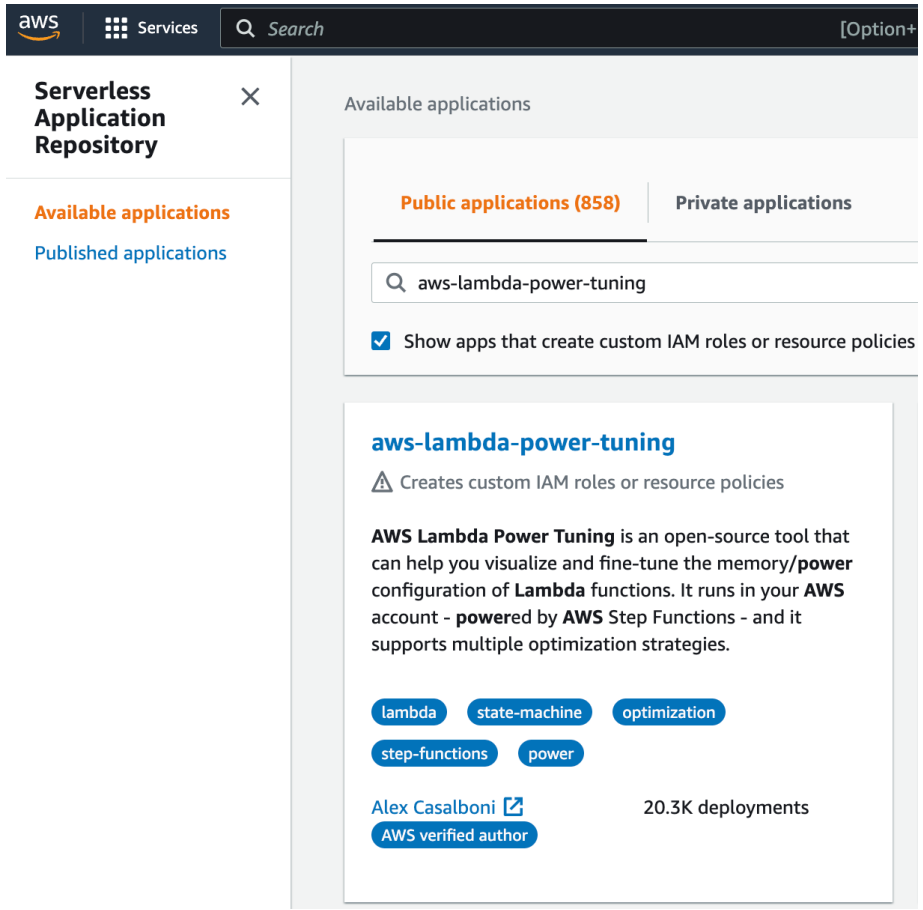
There are numerous ways to resolve cold starts.

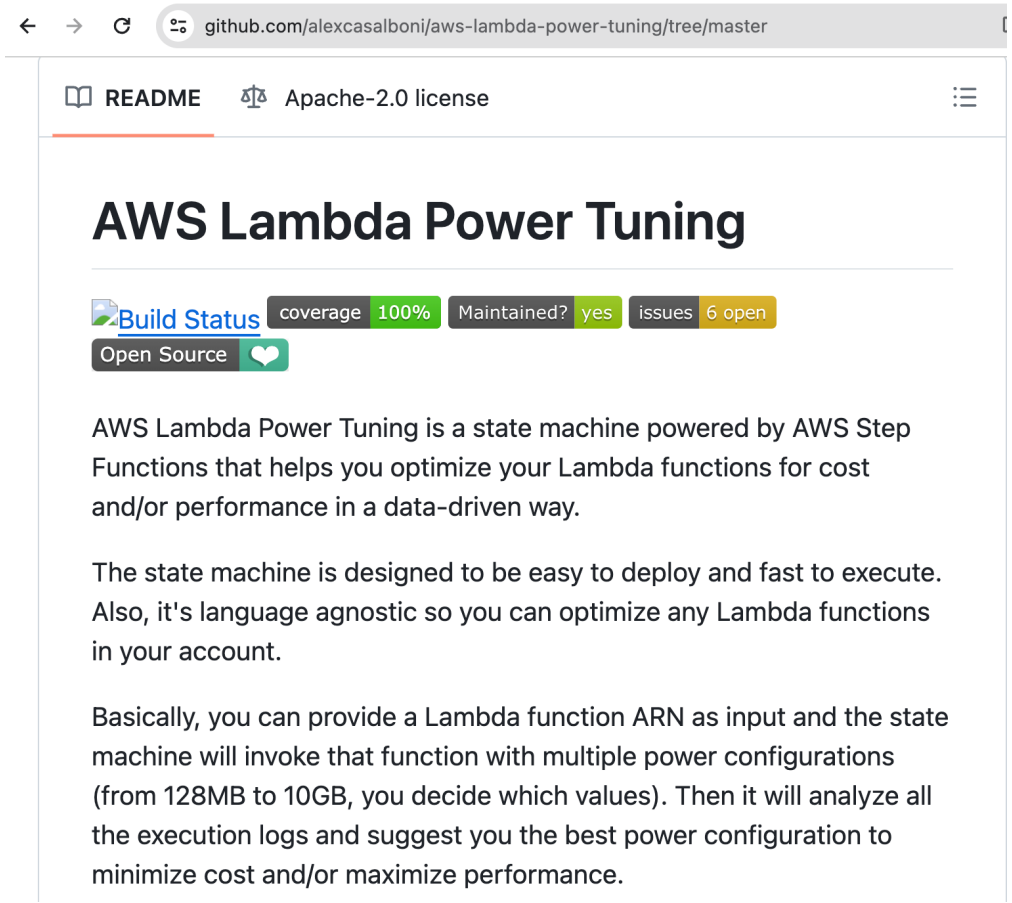
If you need spiky regular throughput with a fast response time, then **Provisioned Concurrency** is your solution.

You can now also use Auto-Scaling to automatically adjust the provisioned concurrency using CloudWatch Metrics!

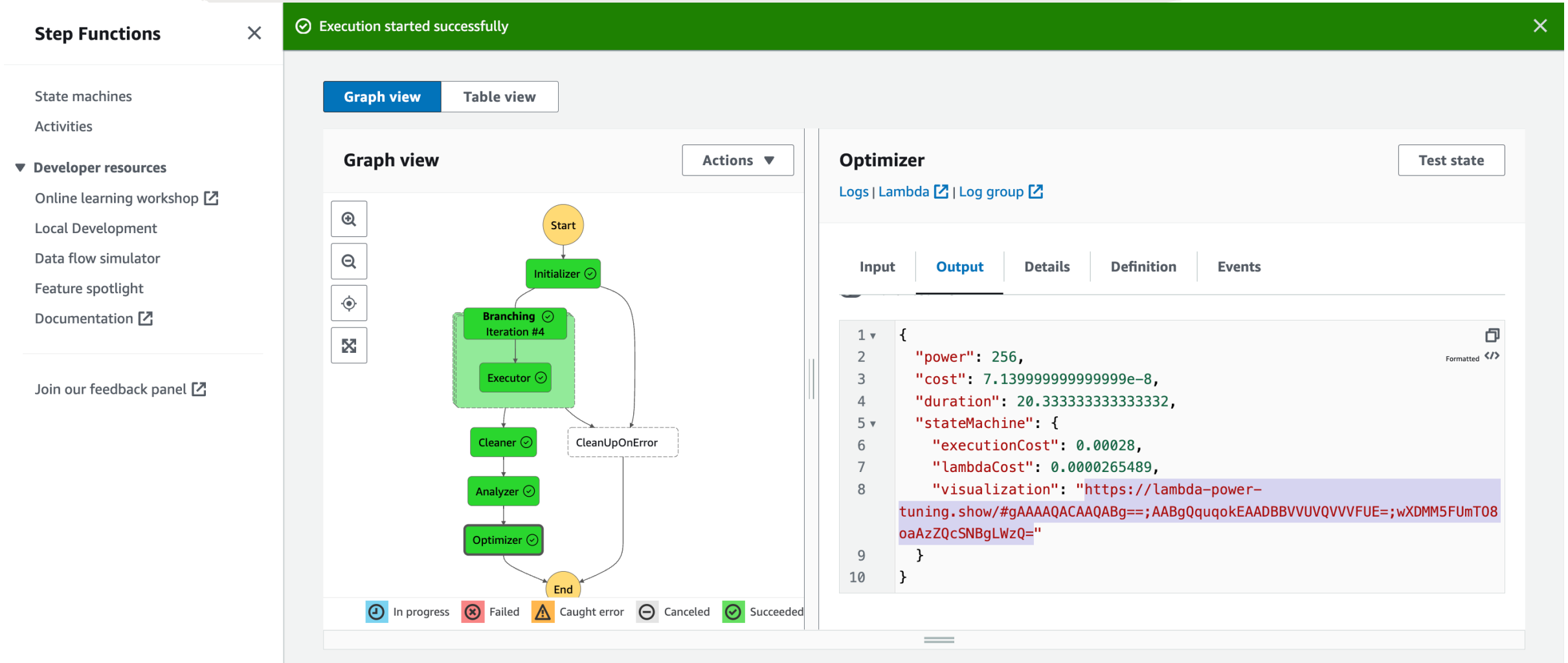


AWS Lambda Power Tuning



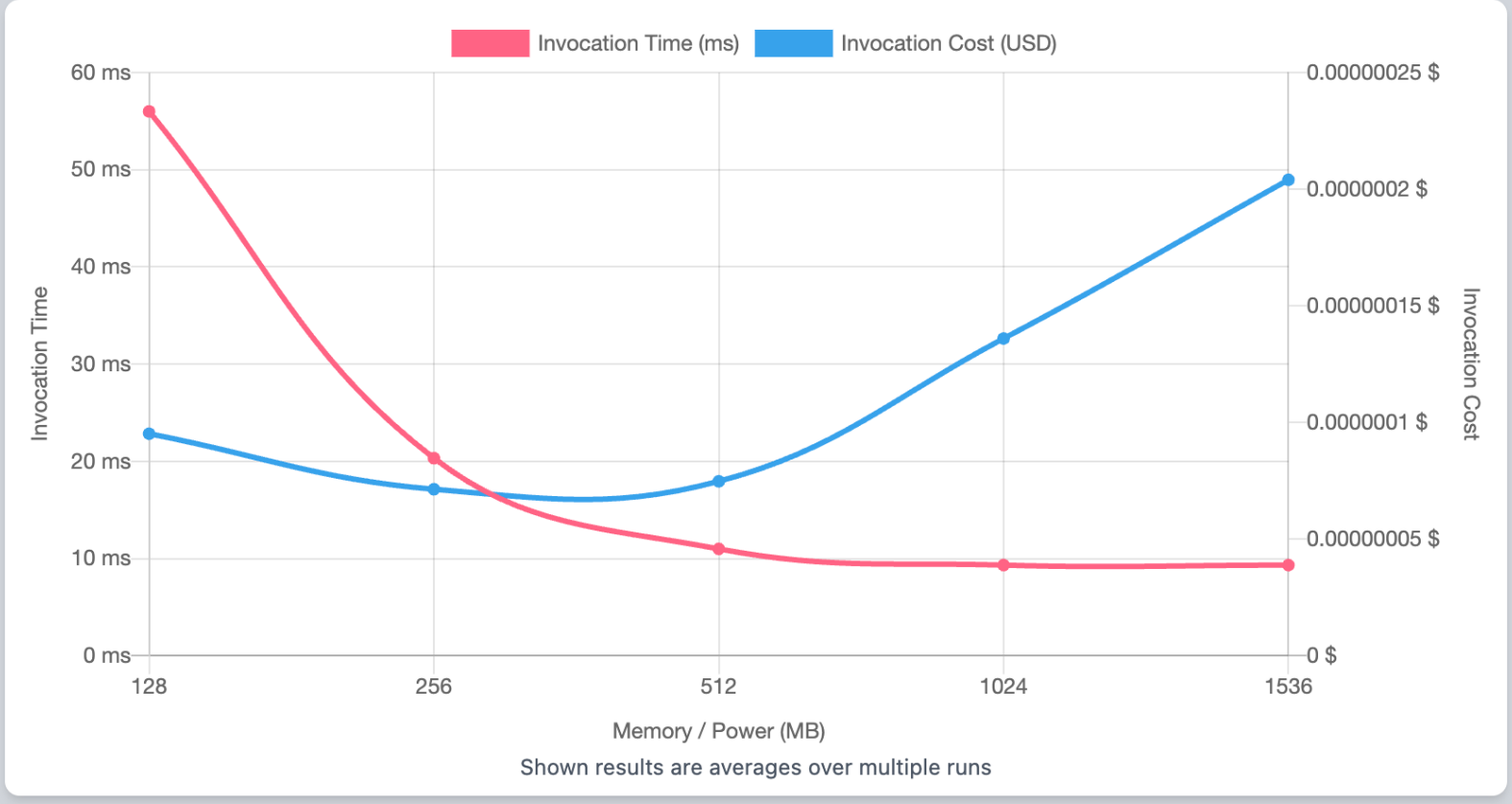


Power Tuning



Power Tuning

AWS Lambda Power Tuning Results



Best Cost
256MB

Best Time
1024MB

Worst Cost
1536MB

Worst Time
128MB

Compare

Code Optimisation

src > adapters > secondary > customer > TS get-customer.adapter.ts > getCustomer

```ts
import { Customer } from '@models/customer';
import { connect } from '@shared/databases-services/customer-service/connection';
import { customerService } from '@shared/databases-services/customer-service/customer/services';

export const getCustomer = async (id: string): Promise<Customer> => {
    console.log('Secondary Adapter: Connecting to database');
    await connect();

    console.log('Secondary Adapter: Retrieving customer Id');
    const { _id, ...customer } = (
        await customerService.getCustomer(id)
    ).toObject();

    return customer as Customer;
};

```

Code Optimisation

src > shared > databases-services > customer-service > connection > TS connect.ts > ...

```ts
import { connect as mongooseConnect, ConnectOptions, Mongoose } from 'mongoose';
import { fetchConnectionSecret } from './fetch-connection-secret';
import { config } from '@config/config';

let connection: Mongoose;

const secretName = config.get('counterDbConnectionSecret');

const defaultConnOptions: ConnectOptions = {
    // Options such as pool size, ssl etc
};

export async function connect(options: ConnectOptions = {}) {
    if (connection) {
        return connection;
    }

    console.log('DB Service: Fetching Connection String');
    console.log(`DB Service: **** ${secretName}`);
    const dbConnectionString = await fetchConnectionSecret(secretName);

    console.log('DB Service: Connecting to database');
    connection = await mongooseConnect(dbConnectionString, {
        ...defaultConnOptions,
        ...options,
    });
    console.log('DB Service: Connected');

    return connection;
}
```

Code Optimisation

src > shared > databases-services > customer-service > connection > TS connect.ts > ...

```ts
import { connect as mongooseConnect, ConnectOptions, Mongoose } from 'mongoose';
import { fetchConnectionSecret } from './fetch-connection-secret';
import { config } from '@config/config';

let connection: Mongoose;

const secretName = config.get('counterDbConnectionSecret');

const defaultConnOptions: ConnectOptions = {
    // Options such as pool size, ssl etc
};

export async function connect(options: ConnectOptions = {}) {
    if (connection) {
        return connection;
    }

    console.log('DB Service: Fetching Connection String');
    console.log(`DB Service: **** ${secretName}`);
    const dbConnectionString = await fetchConnectionSecret(secretName);

    console.log('DB Service: Connecting to database');
    connection = await mongooseConnect(dbConnectionString, {
        ...defaultConnOptions,
        ...options,
    });
    console.log('DB Service: Connected');

    return connection;
}
```

Code Optimisation

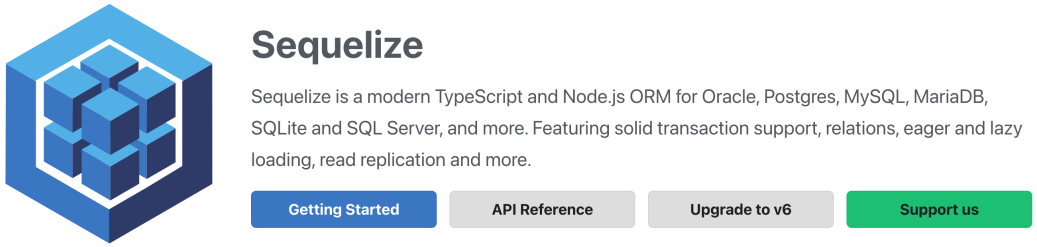
src > shared > databases-services > customer-service > connection > TS fetch-connection-

```typescript
import {
    GetSecretValueCommand,
    SecretsManagerClient,
} from '@aws-sdk/client-secrets-manager';

const client = new SecretsManagerClient();

interface ConnectionSecret {
    dbConnectionString: string;
}

export const fetchConnectionSecret = async (
    secretName: string
): Promise<string> => {
    const response = await client.send(
        new GetSecretValueCommand({
            SecretId: secretName,
        })
    );
    const connectionSecret: ConnectionSecret = JSON.parse(
        response.SecretString || ''
    );

    if (!connectionSecret || !connectionSecret.dbConnectionString) {
        throw new Error('Unable to parse connection details');
    }

    return connectionSecret.dbConnectionString;
};
```

Code Optimisation

Also, Avoid sizable packages where possible.

Such as Sequelize ORM
https://www.npmjs.com/package/sequelize



Performance – Throughput

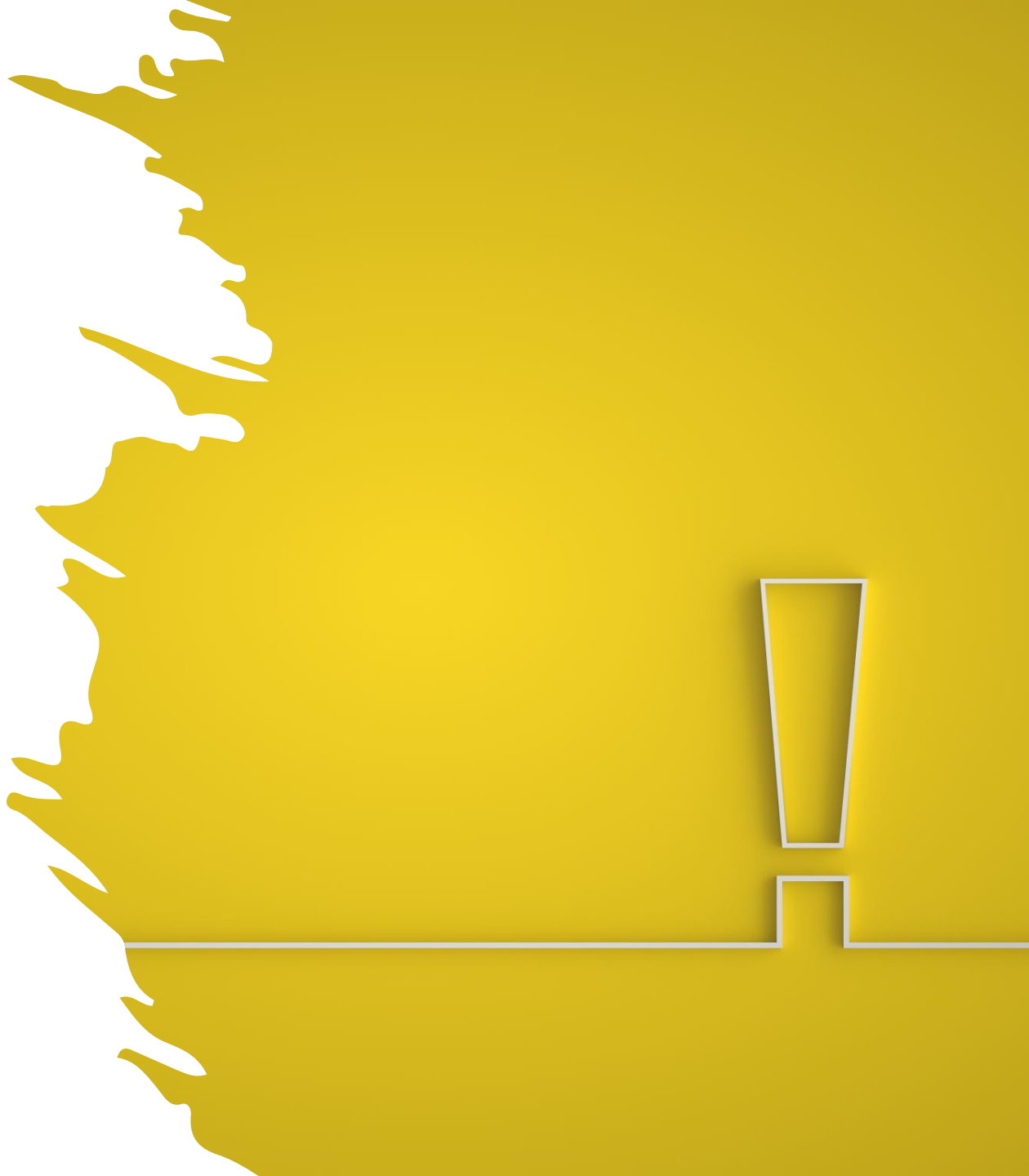
How many concurrent users are likely to use this application. We could have 100 users, 500 users or 10,000 users.

For Throughput -

Number of active users VS response time.

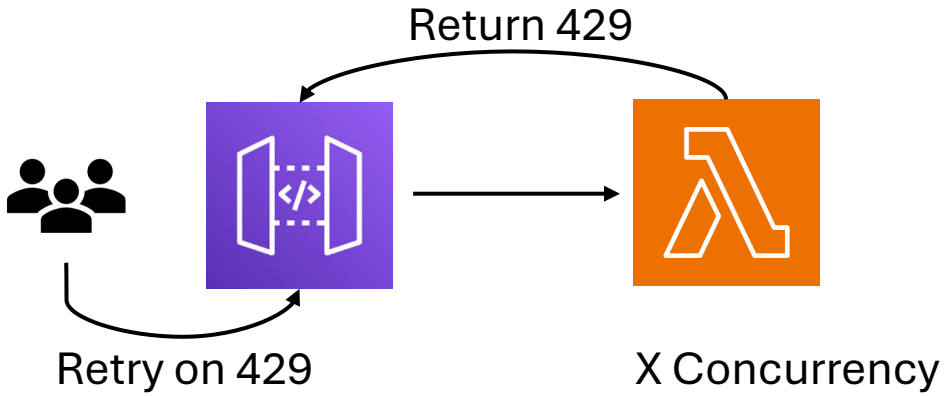
We can then use both to generate a baseline.

For example, the system needs to manage: *100 requests per second*.



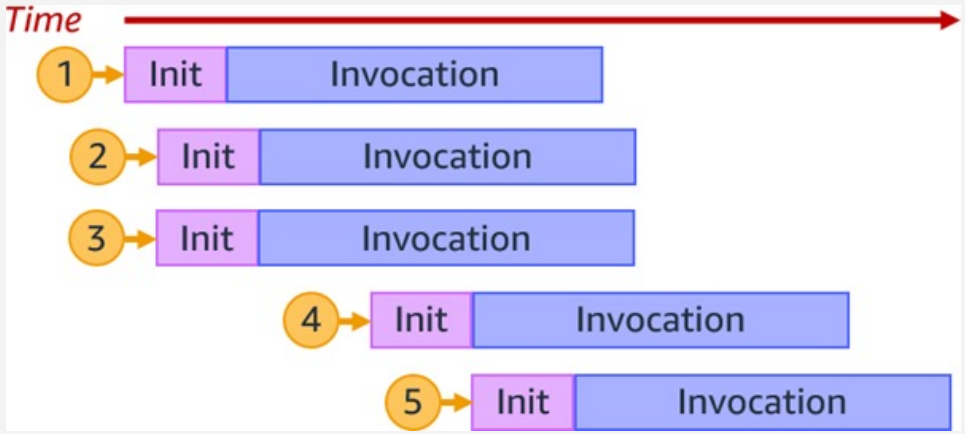
Performance –

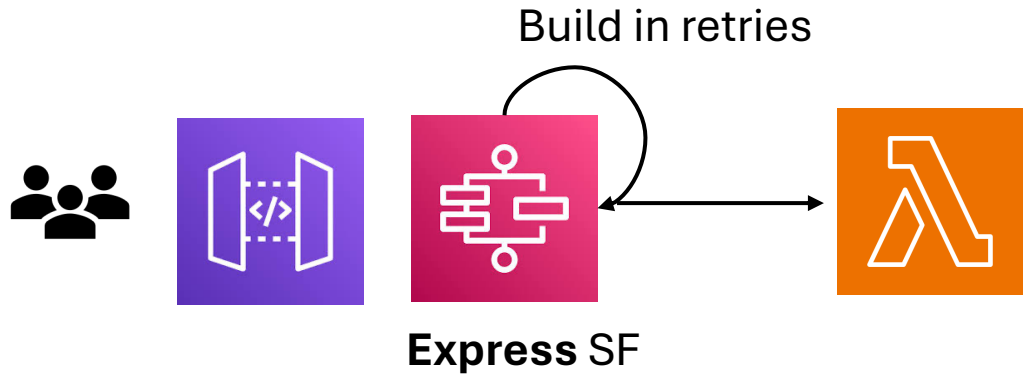
- **Managing throughput with lambda throttling**



Synchronously

- Concurrent Executions - Default 1,000 per account – Soft Limit, up to tens of thousands





https://aws.amazon.com/blogs/compute/understanding-aws-lambda-scaling-and-throughput/ - Julian Wood

Performance –

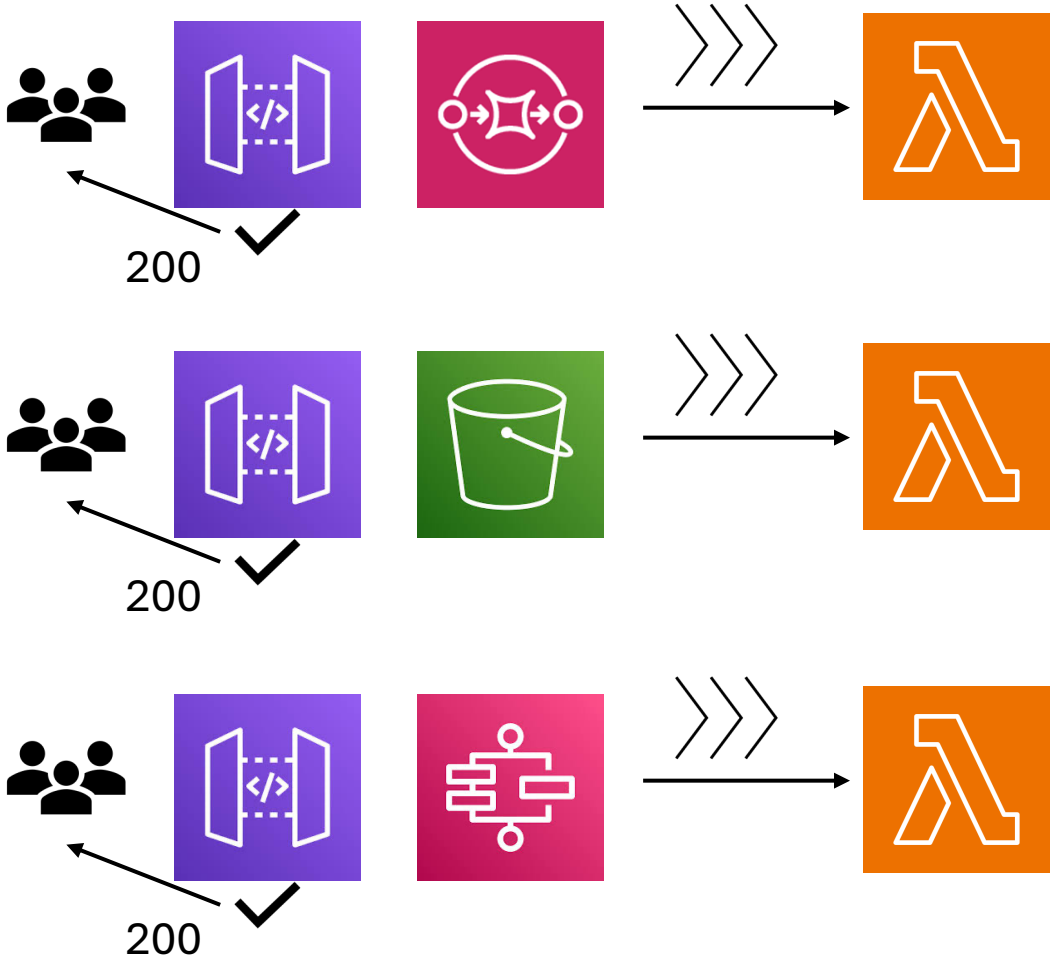
- **Throttling our lambda input**

Asynchronously

- Concurrent Executions - Default 1,000 per account – Soft Limit, up to tens of thousands

Event Driven

Storage First



Scalability

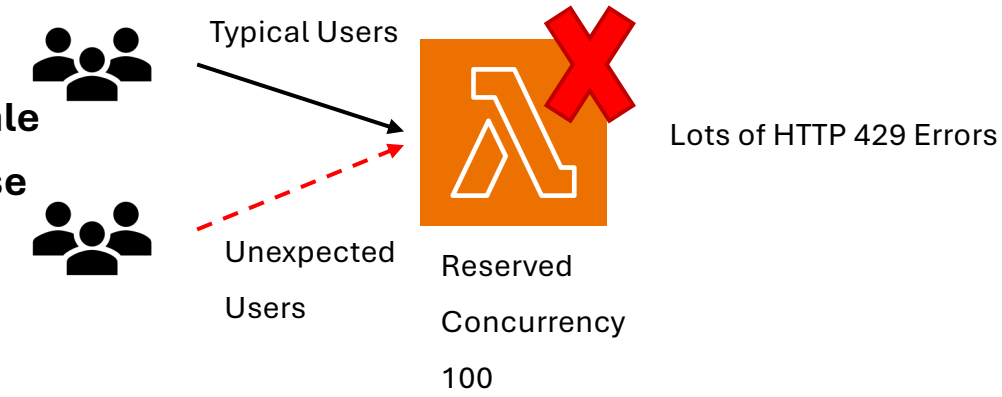
Scalability is often measured with how the system reacts to an increase in load.

Typically, we load test our application so we are confident it will scale in production.

How many concurrent users or actions does the system need to support? What spikes will occur?

Special Events

- **Black Friday Sale**
- **Product Release**
- **Payday**
- **Advertising**



Scalability

Scalability is often left behind and **taken for granted** with serverless applications. Engineers typically say, “lambda can scale indefinitely”, “DynamoDB will manage any throughput”.

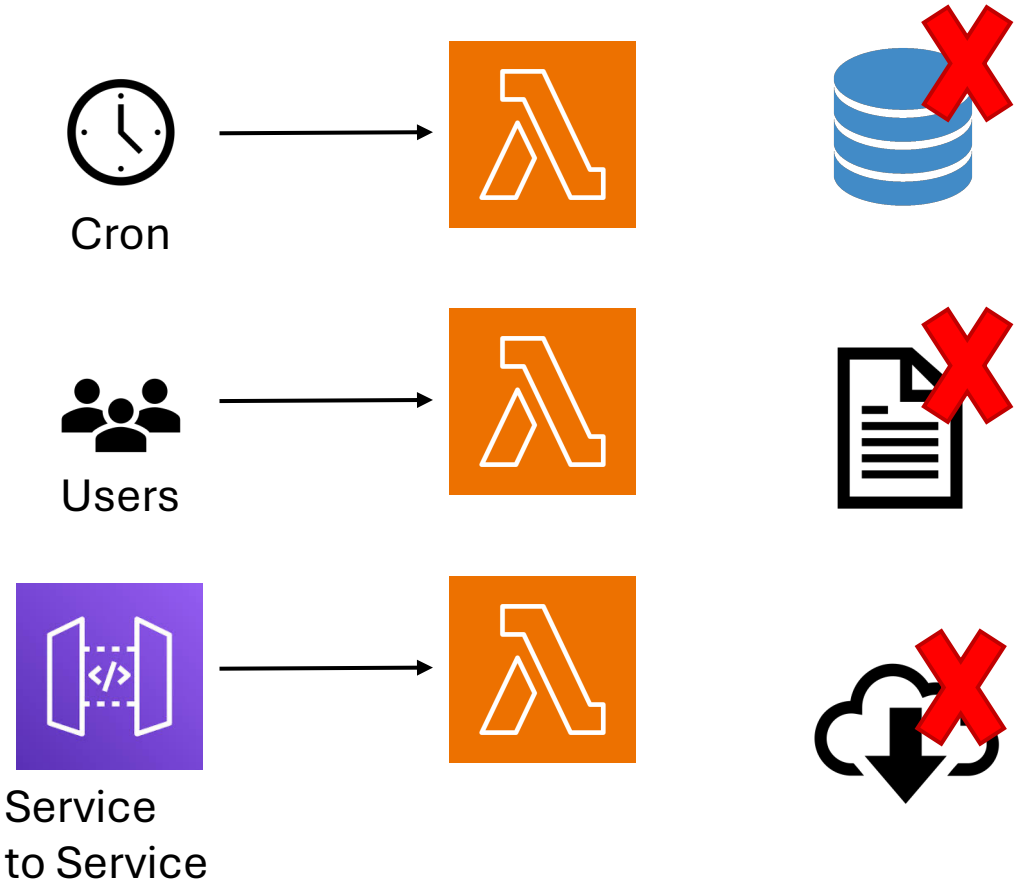
Well not without complications… scaling up for these services takes time, and your application will receive scalability errors (HTTP 429 -Too Many Requests) during the scaling out (horizontal scaling) period.

You may also depend on another service, or read from a file, call a downstream API.

All of these instances can cause your serverless applications to fail scalability checks.

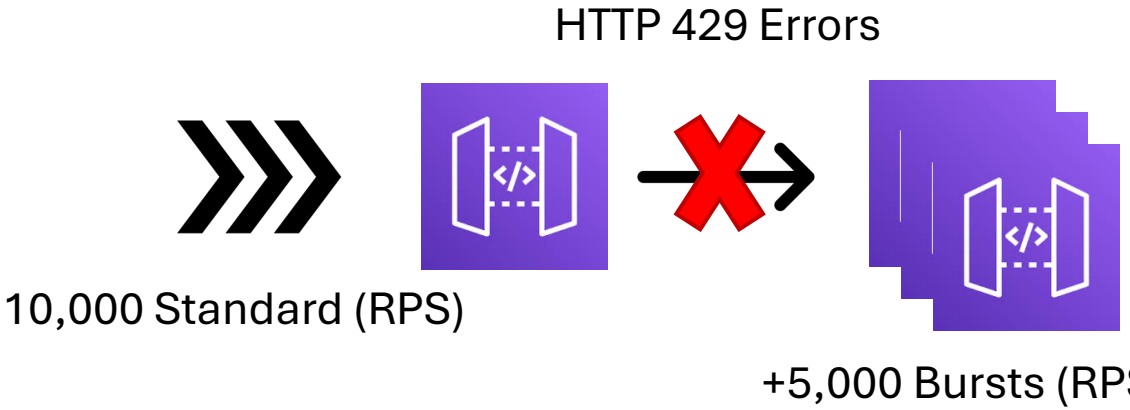
Scalability – Common Problems

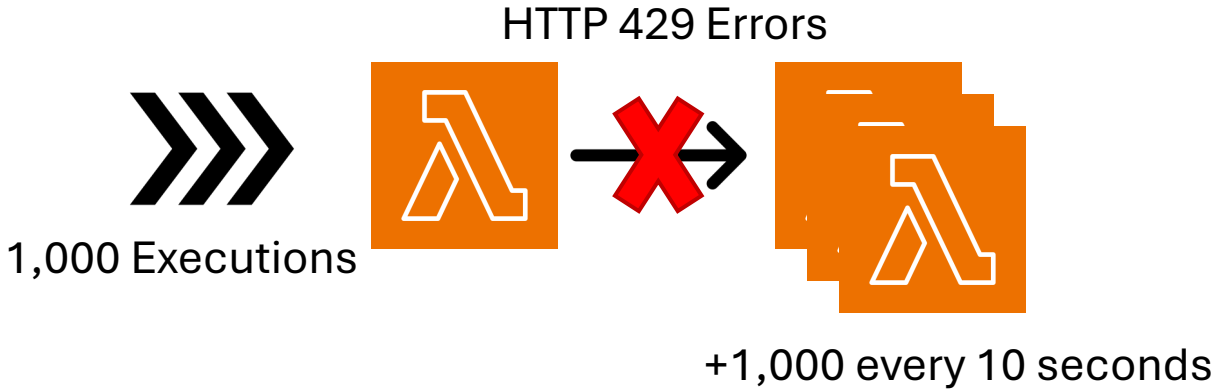
- The dependency bottleneck
  - Database connections hit
  - File limits reached
  - Capability of External APIs

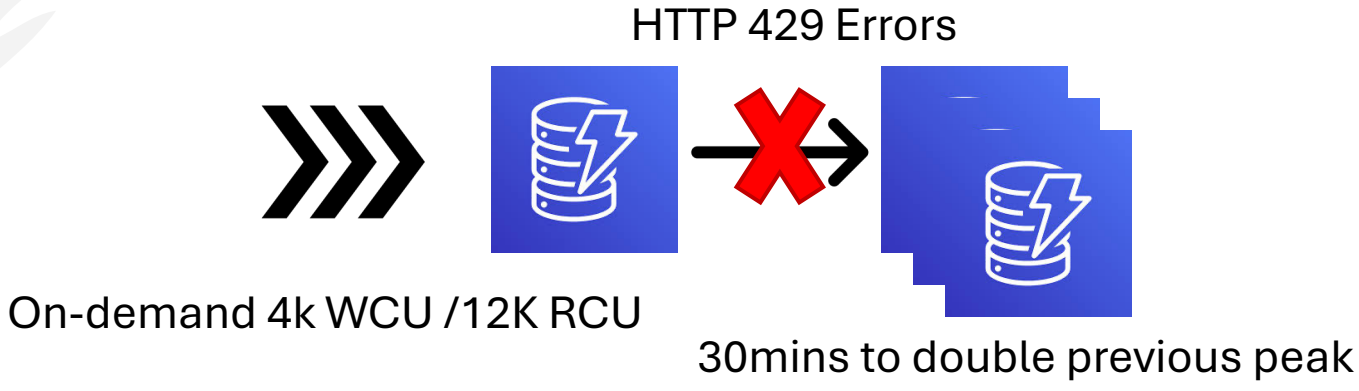


Scalability – Often taken for granted...

- **Service Scaling**
  - **API Gateway**
  - **DynamoDB**
  - **Lambda**







Scalability/ Throughput

Remember we mentioned Scalability with Throughput...

To scale safely, a well-designed system should be implementing at minimum some exception handling including basic back-off and retry.

```
22  async function getCustomerCall(id: string) {
23      try {
24          return await customerService.getCustomer(id);
25      } catch (error) {
26          // Advanced - Check error code. i.e 429
27          if (attempt < 3) {
28              attempt += 1;
29              const backoffDelay = 100;
30              console.warn(
31                  `Request failed. Retry attempt ${attempt} after ${backoffDelay}ms`
32              );
33              await new Promise((resolve) => setTimeout(resolve, backoffDelay));
34              return getCustomerCall(id);
35          }
36      }
37      throw error;
38  }
```

In Summary

It’s often easy to miss off Non-Functional Requirements, especially when teams are under pressure of delivery.

They are critical to the success of a software system because they define the system's operational qualities and constraints, ensuring it meets user needs and performs well under expected conditions.

Ignoring NFRs can lead to significant issues post-deployment, such as system crashes, security breaches, poor user satisfaction, and costly maintenance.

Properly defining and addressing NFRs from the beginning ensures a robust, secure, and efficient system that meets user expectations and performs well in real-world conditions.