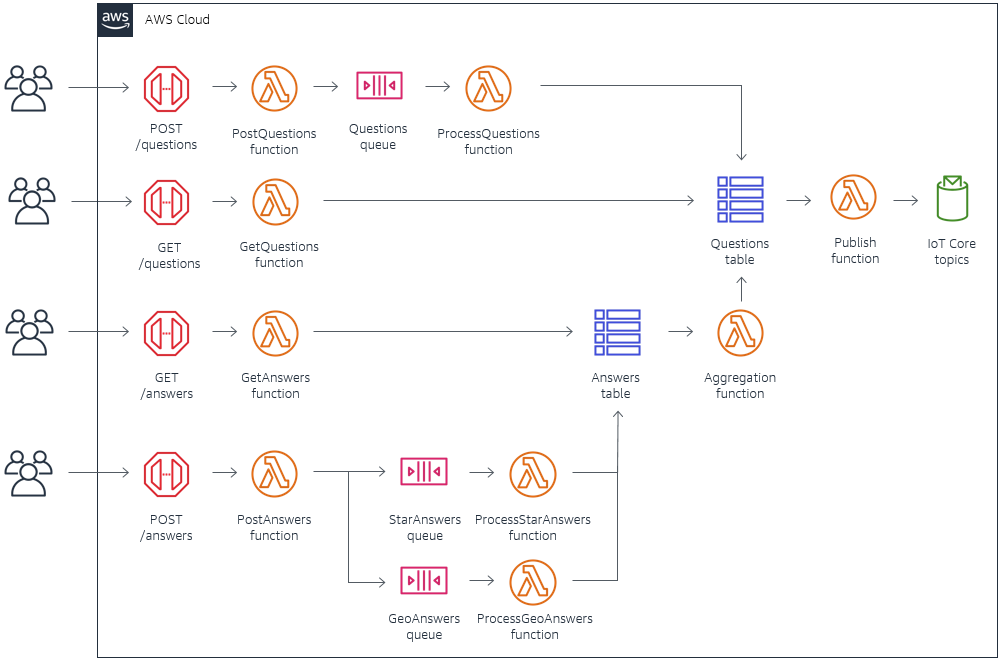
Load testing a web application’s serverless backend

by [James Beswick](https://aws.amazon.com/blogs/compute/author/jbeswick/) | on 06 JUL 2020 | in [Amazon API Gateway](https://aws.amazon.com/blogs/compute/category/application-services/amazon-api-gateway-application-services/), [Amazon CloudWatch](https://aws.amazon.com/blogs/compute/category/management-tools/amazon-cloudwatch/), [Amazon DynamoDB](https://aws.amazon.com/blogs/compute/category/database/amazon-dynamodb/), [AWS Cloud9](https://aws.amazon.com/blogs/compute/category/developer-tools/aws-cloud9/), [AWS Lambda](https://aws.amazon.com/blogs/compute/category/compute/aws-lambda/), [Best Practices](https://aws.amazon.com/blogs/compute/category/post-types/best-practices/), [Serverless](https://aws.amazon.com/blogs/compute/category/serverless/), [Technical How-To](https://aws.amazon.com/blogs/compute/category/post-types/technical-how-to/) | [Permalink](https://aws.amazon.com/blogs/compute/load-testing-a-web-applications-serverless-backend/) | [Comments](https://commenting.awsblogs.com/embed.html?disqus_shortname=aws-compute&disqus_identifier=10369&disqus_title=Load+testing+a+web+application%E2%80%99s+serverless+backend&disqus_url=https://aws.amazon.com/blogs/compute/load-testing-a-web-applications-serverless-backend/) | [Share](https://aws.amazon.com/blogs/compute/load-testing-a-web-applications-serverless-backend/)

Many web applications experience high levels of traffic and spiky load patterns. The goal of load testing is to ensure that the architecture and system design works for the amount of traffic expected. It can help developers find bottlenecks and unexpected behavior before a system is deployed to production. This post uses the [*Ask Around Me*application](https://github.com/aws-samples/ask-around-me) as an example to show how to test load in a serverless architecture.

In *Ask Around Me*, users ask and answer questions in their local geographic area. The expected hourly load is 1,000 new questions, 10,000 new answers, and 50,000 question lookup queries. I use these numbers as a baseline for the tests. This is the architecture of the *Ask Around Me* backend application:

[](https://d2908q01vomqb2.cloudfront.net/1b6453892473a467d07372d45eb05abc2031647a/2020/07/01/load-testing1.png)

**Focus areas for load testing**

In serverless architectures using AWS services, you can perform a round-trip test from an API endpoint. You can also isolate areas in the design where you should test performance. API testing provides the best approximation of the performance that users experience but it may not always be possible. You can also isolate microservices consuming from SQS queue or receive events from [Amazon EventBridge](https://aws.amazon.com/eventbridge/), and test only those parts of the infrastructure.

While AWS services are built to withstand high levels of traffic, it’s important to consider the effect of Service Quotas on your application. Service Quotas are applied at the Region and account levels depending upon the service. You can view all your quotas in one place from the [Service Quotas console](https://console.aws.amazon.com/servicequotas/home?region=us-east-1#!/). These are designed to protect you and other customers if your applications use more resources than planned. These quotas consist of hard and soft limits. For soft limits, you can request quota increases by [opening a support ticket](https://console.aws.amazon.com/support/home?region=us-east-1#/case/create).

You must also consider downstream services. While serverless services like Lambda scale on your behalf, you may use downstream services that could be overwhelmed when traffic increases. Load testing can help identify these areas. You can implement mechanisms like queuing, caching, or pooling to protect those non-serverless parts of your infrastructure. If you are using [Amazon RDS](https://aws.amazon.com/rds/), for example, you might implement [Amazon RDS Proxy](https://aws.amazon.com/rds/proxy/) to help pool and scale resources.

Finally, load testing can help identify custom code in Lambda functions that may not run efficiently as traffic scales up. Typically, these issues are caused by the code itself or the function configuration. For example, code may process event batches effectively or may not be configured with the [appropriate concurrency](https://docs.aws.amazon.com/lambda/latest/dg/configuration-concurrency.html) or [memory configuration](https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html). Frequently these issues are unnoticed in development but resurface in a load test.

**Load testing tools**

Load testing serverless infrastructure can be both inexpensive and systematic. There are several tools available for serverless developers to perform this task. One of the most popular is [Artillery Community Edition](https://artillery.io/), which is an open-source tool for testing serverless APIs. You configure the number of requests per second and overall test duration, and it uses a headless Chromium browser to run its test flows.

The performance report measures the roundtrip time from the client device, so can be affected by your machine’s performance and network. One way to eliminate your local network’s impact on the results is to use [AWS Cloud9](https://aws.amazon.com/cloud9/) to run the tests remotely.

For Artillery, the maximum number of concurrent tests is constrained by your local computing resources and network. To achieve higher throughput, you can use [Serverless Artillery](https://www.npmjs.com/package/serverless-artillery), which runs the Artillery package on Lambda functions. As a result, this tool can scale up to a significantly higher number of tests.

The *Ask Around Me* application is deployed in my AWS account – see the [application’s blog series](https://aws.amazon.com/blogs/compute/building-a-location-based-scalable-serverless-web-app-part-1/) to learn more about the deployment process. I use an [AWS Cloud9](https://aws.amazon.com/cloud9/) instance to run these API tests:

1. Adding 1,000 questions per hour using the POST /questions API.
2. Adding 10,000 answers per hour using the POST /answers API.
3. Fetching 50,000 questions per hour based upon random geo-location using the GET /questions API.

You can find the test scripts and Artillery configurations in the *testing* directory of the application’s [GitHub repo](https://github.com/aws-samples/ask-around-me/tree/master/backend/loadTesting).

Artillery also enables you to specify custom functions to provide randomized data and custom query parameters, as required by your API. The loadTestFunction.js file contains a function to return randomized geo-point and rating data per test:

// Sets a bounding box around an area in Virginia, USA

const bounds = {

latMax: 38.735083,

latMin: 40.898677,

lngMax: -77.109339,

lngMin: -81.587841

}

const generateRandomData = (userContext, events, done) => {

const randomLat = ((bounds.latMax-bounds.latMin) \* Math.random()) + bounds.latMin

const randomLng = ((bounds.lngMax-bounds.lngMin) \* Math.random()) + bounds.lngMin

const id = parseInt(Math.random()\*1000000)+1 //random 0-1000000

const rating = parseInt(Math.random()\*5)+1 //returns 1-5

userContext.vars.lat = randomLat.toFixed(7)

userContext.vars.lng = randomLng.toFixed(7)

userContext.vars.id = id

userContext.vars.rating = rating

return done()

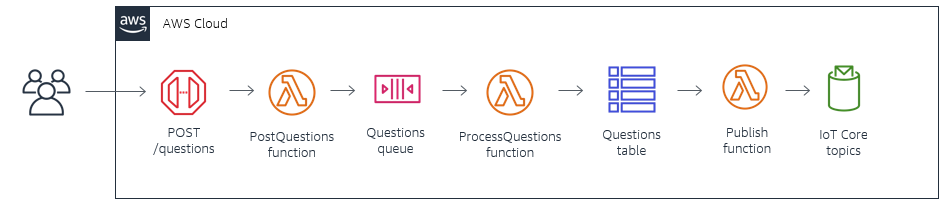
}

module.exports = { generateRandomData }

JavaScript

**Test #1: Adding 1,000 questions per hour**

The POST *questions* API has the following architecture:

[](https://d2908q01vomqb2.cloudfront.net/1b6453892473a467d07372d45eb05abc2031647a/2020/07/01/load-testing2.png)

The Artillery configuration file [1-*test.yaml*](https://github.com/aws-samples/ask-around-me/blob/master/backend/loadTesting/1-questions/1-test.yaml) is set to create three requests per second over a 5-minute duration. This equates to 10,800 questions per hour, significantly higher than the estimated load for this function. The scenario specifies the JSON payload expected by the *questions* API:

config:

target: 'https://abcd1234567.execute-api.us-east-1.amazonaws.com'

phases:

- duration: 300

arrivalRate: 3

processor: "./loadTestFunction.js"

defaults:

headers:

Authorization: 'Bearer <<enter your valid JWT token>>'

scenarios:

- flow:

- function: "generateRandomData"

- post:

url: "/questions"

json:

question: "This is a load test question - #{{ id }}"

type: "Star rating"

position:

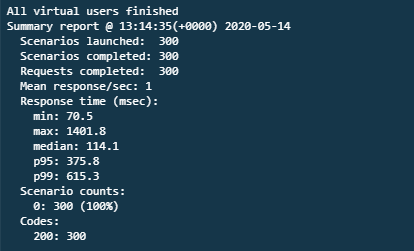
latitude: {{ lat }}

longitude: {{ lng }}

- log: "Sent POST request to / with {{ lat }}, {{ lng }}"

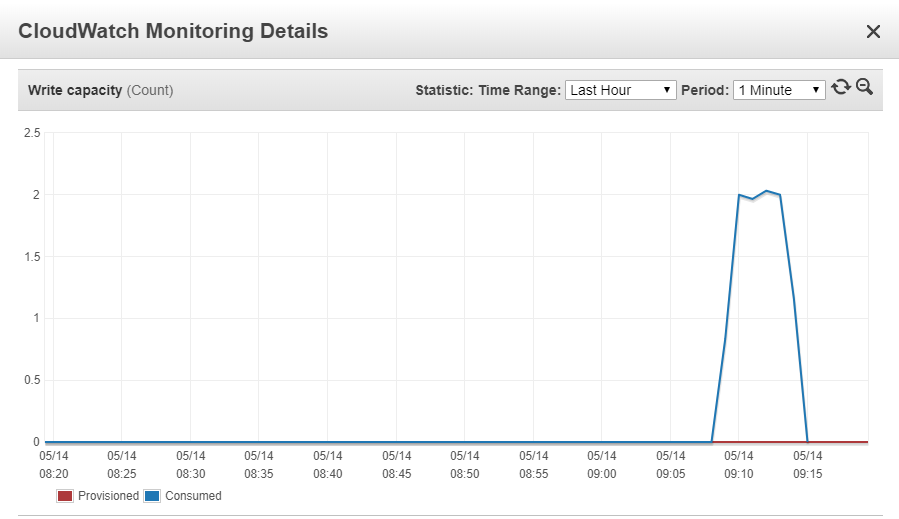
YAML

You execute the Artillery test with the command artillery run ./1-test.yaml. My test concludes with the following results:

[](https://d2908q01vomqb2.cloudfront.net/1b6453892473a467d07372d45eb05abc2031647a/2020/07/01/load-testing3.png)

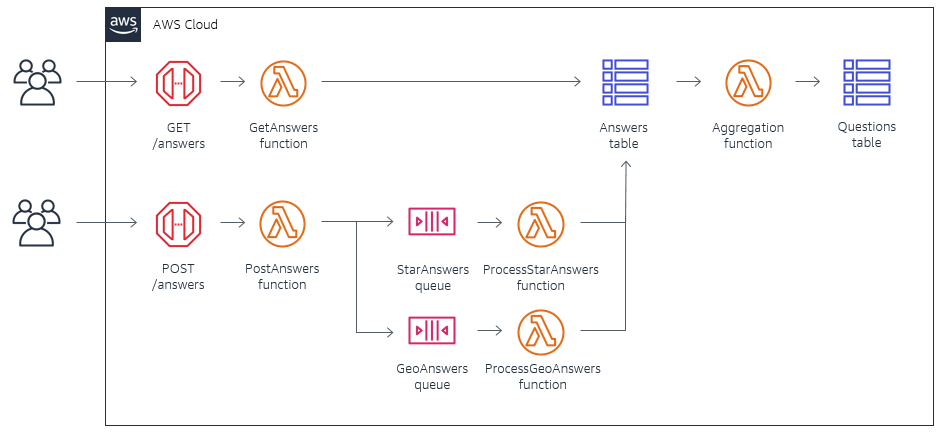
Over 300 requests, the median response time is 114 ms. The p95 response time shows that 95% of all responses are served within 376 ms. The slowest response of 1401 ms is caused by cold starts when the Lambda service scales up the underlying function due to load.

As this process writes to a DynamoDB table, I can also see how many write capacity units (WCUs) are consumed by the test. From the [DynamoDB console](https://console.aws.amazon.com/dynamodb/home), select the table *aamQuestions*, then choose the Metrics tab. This shows the *Write capacity*metric:

[](https://d2908q01vomqb2.cloudfront.net/1b6453892473a467d07372d45eb05abc2031647a/2020/07/01/load-testing4.png)

**Test #2: Adding 10,000 answers per hour.**

The POST *answers* API has the following architecture:

[](https://d2908q01vomqb2.cloudfront.net/1b6453892473a467d07372d45eb05abc2031647a/2020/07/01/load-testing5.png)

The Artillery configuration in [2-test.yaml](https://github.com/aws-samples/ask-around-me/blob/master/backend/loadTesting/2-answers/2-test.yaml) creates 10 answers per second over a 5-minute duration. This equates to 36,000 per hour, much higher than the estimated load. The scenario defines the randomized rating used by the testing process:

config:

target: 'https://abcd1234567.execute-api.us-east-1.amazonaws.com'

phases:

- duration: 300

arrivalRate: 10

processor: "./loadTestFunction.js"

defaults:

headers:

Authorization: 'Bearer <<enter your valid JWT token>>’

scenarios:

- flow:

- function: "generateRandomData"

- post:

url: "/answers"

json:

type: "Star"

rating: "{{ rating }}"

question:

type: "Star"

latitude: 39.08259127440097

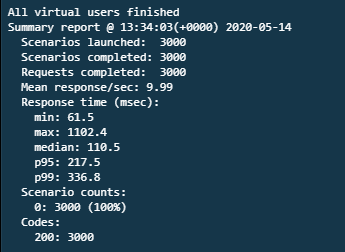
longitude: -77.46246339003038

rangeKey: "testuser|1-1589380702281"

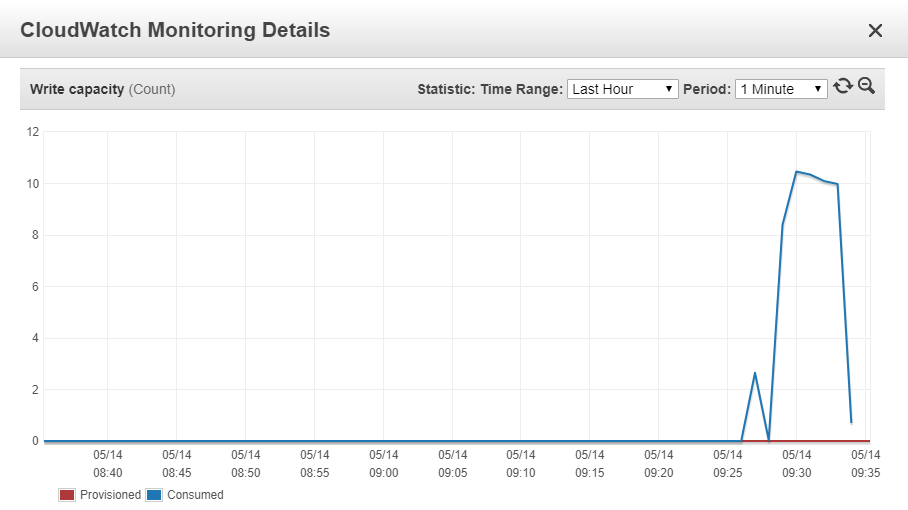
- log: "Sent POST request to / with {{ rating }}"

YAML

The test results show a median response time of 111 ms with a p95 time of 218 ms. In the worst case, a request took 1102 ms to complete:

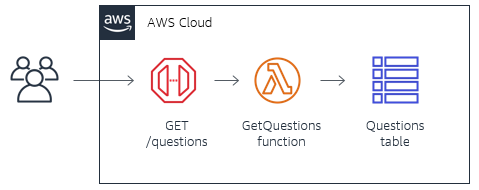
[](https://d2908q01vomqb2.cloudfront.net/1b6453892473a467d07372d45eb05abc2031647a/2020/07/01/load-testing6.png)

Checking the *Metrics* tab for the *aaAnswers* table, this test consumed just under 11 WCUs at peak:

[](https://d2908q01vomqb2.cloudfront.net/1b6453892473a467d07372d45eb05abc2031647a/2020/07/01/load-testing7.png)

**Test #3: Fetching 50,000 questions per hour**

The GET *questions* API invokes a Lambda function that uses the [Geo Library for Amazon DynamoDB](https://www.npmjs.com/package/dynamodb-geo):

[](https://d2908q01vomqb2.cloudfront.net/1b6453892473a467d07372d45eb05abc2031647a/2020/07/01/load-testing8.png)

This process is read-intensive on the underlying DynamoDB table. The [testing configuration](https://github.com/aws-samples/ask-around-me/blob/master/backend/loadTesting/3-geo-test/3-test.yaml) simulates 20 queries per second over 2 minutes for random locations in a bounding box around Virginia, USA:

config:

target: 'https://abcd1234567.execute-api.us-east-1.amazonaws.com'

phases:

- duration: 120

arrivalRate: 20

processor: "./loadTestFunction.js"

defaults:

headers:

Authorization: 'Bearer <<enter your valid JWT token>>’

scenarios:

- flow:

- function: "generateRandomData"

- get:

url: "/questions"

qs:

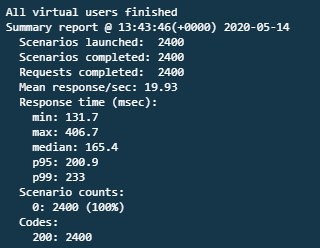
lat: "{{ lat }}"

lng: "{{ lng }}"

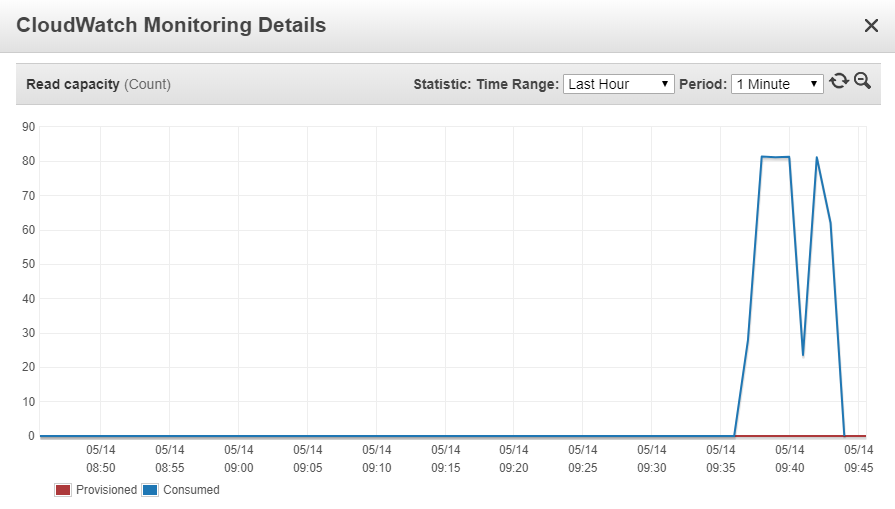
- log: "Sent POST request to / with {{ lat }}, {{ lng }}"

YAML

This is a synchronous API so the performance directly impacts the user’s experience of the application. This test shows that the median response time is 165 ms with a p95 time of 201 ms:

[](https://d2908q01vomqb2.cloudfront.net/1b6453892473a467d07372d45eb05abc2031647a/2020/07/01/load-testing9.png)

This level of load equates to 72,000 queries per hour, almost 50% above the expected usage. The DynamoDB metrics show a peak consumption of 82 read capacity units:

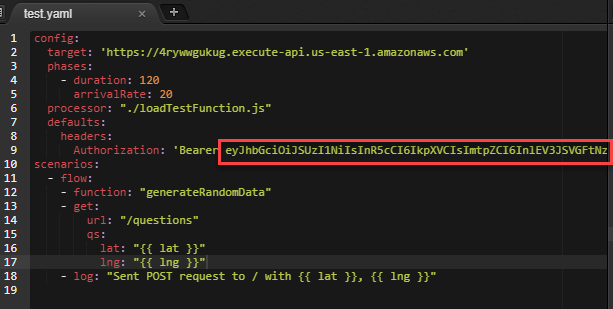
[](https://d2908q01vomqb2.cloudfront.net/1b6453892473a467d07372d45eb05abc2031647a/2020/07/01/load-testing10.png)

**Testing authenticated routes**

These API routes are protected from public access and require authorization. This application uses HTTP APIs, which accepts JWT tokens, and it uses Auth0 in the frontend application to generate these tokens. When you are load testing API Gateway routes with custom authorizers, you have a number of options.

At the early development stage, you may choose to remove the authentication to perform load tests. This simplifies the process but is not recommended beyond research and prototyping. If you turn off authentication for testing, there is a risk that it is not enabled again for production. This would leave your routes open to the public.

A better approach is to create a test user in your identity provider and use the JWT token for testing. Auth0 allows you to [obtain a token manually](https://auth0.com/docs/api/management/v2/get-access-tokens-for-test), and use this in the Artillery configuration for the authorization header:

[](https://d2908q01vomqb2.cloudfront.net/1b6453892473a467d07372d45eb05abc2031647a/2020/07/01/load-testing11.png)

Since custom code frequently uses the decoded identity in processing, supplying a test token provides the closest simulation of actual usage. You must refresh this token in the test scripts periodically, and you can change [scopes](https://auth0.com/docs/scopes/current) as needed.

The testing directory in the GitHub repo also includes a script for testing functions that consume from SQS queues. This allows you to test microservices further down in your infrastructure stack. This script injects messages into the SQS queue, simulating upstream processes.

**Conclusion**

In this post, I discuss focus areas for load testing of serverless applications, and highlight two tools commonly used. I show how to configure Artillery with customized functions, and how to run tests to simulate load on the *Ask Around Me* application.

I cover some of the options for testing authenticated API Gateway routes and how you can use JWT tokens in your load testing configuration. You can also test microservices within a serverless architecture by injecting messages into SQS queues to simulate upstream load.