AWS Compute Blog

Building well-architected serverless applications: Regulating inbound request rates – part 1

by Julian Wood | on 22 JUL 2021 | in Amazon API Gateway, Amazon Cognito, Amazon DynamoDB, Amazon Simple Notification Service (SNS), Amazon Simple Queue Service (SQS), AWS Amplify, AWS CloudFormation, AWS Lambda, AWS Step Functions, AWS Well-Architected, Kinesis Data Streams, Serverless | Permalink | Share

This series of blog posts uses the AWS Well-Architected Tool with the Serverless Lens to help customers build and operate applications using best practices. In each post, I address the serverless-specific questions identified by the Serverless Lens along with the recommended best practices. See the introduction post for a table of contents and explanation of the example application.

Reliability question REL1: How do you regulate inbound request rates?

> Defining, analyzing, and enforcing inbound request rates helps achieve better throughput. Regulation helps you adapt different scaling mechanisms based on customer demand. By regulating inbound request rates, you can achieve better throughput, and adapt client request submissions to a request rate that your workload can support.

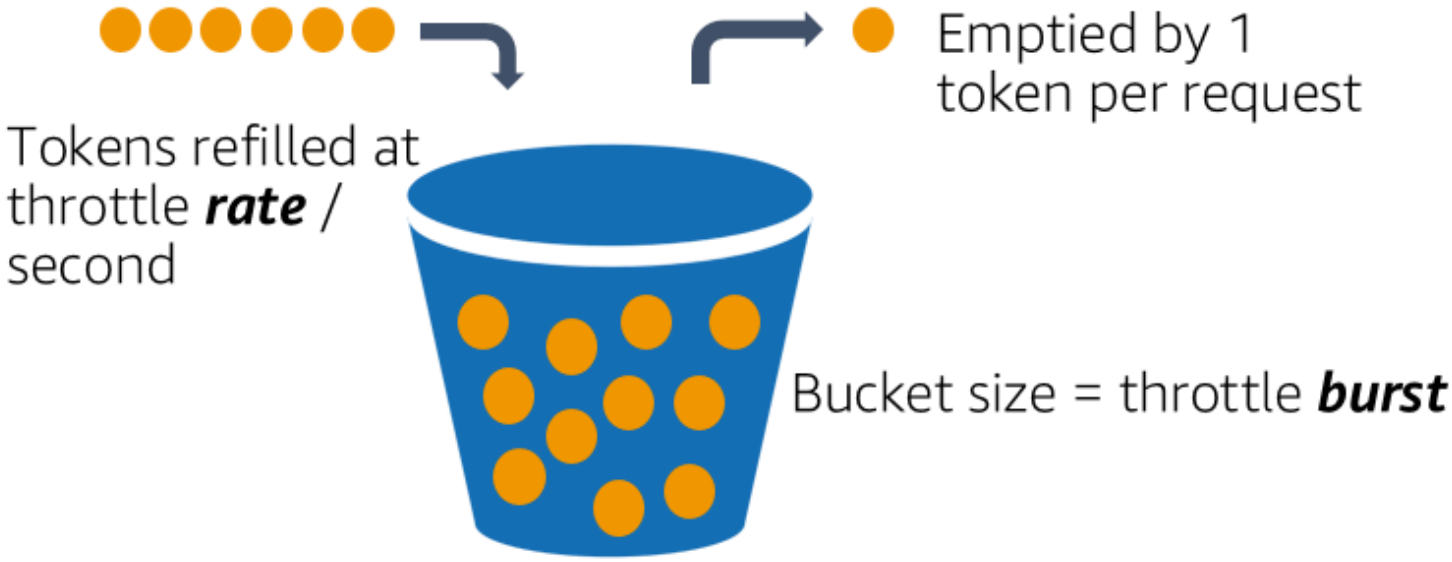
Required practice: Control inbound request rates using throttling

Throttle inbound request rates using steady-rate and burst rate requests

Throttling requests limits the number of requests a client can make during a certain period of time. Throttling allows you to control your API traffic. This helps your backend services maintain their performance and availability levels by limiting the number of requests to actual system throughput.

To prevent your API from being overwhelmed by too many requests, Amazon API Gateway throttles requests to your API. These limits are applied across all clients using the token bucket algorithm. API Gateway sets a limit on a steady-state *rate* and a *burst* of request submissions. The algorithm is based on an analogy of filling and emptying a bucket of tokens representing the number of available requests that can be processed.

Each API request removes a token from the bucket. The throttle *rate* then determines how many requests are allowed per second. The throttle *burst* determines how many concurrent requests are allowed. I explain the token bucket algorithm in more detail in "Building well-architected serverless applications: Controlling serverless API access – part 2"



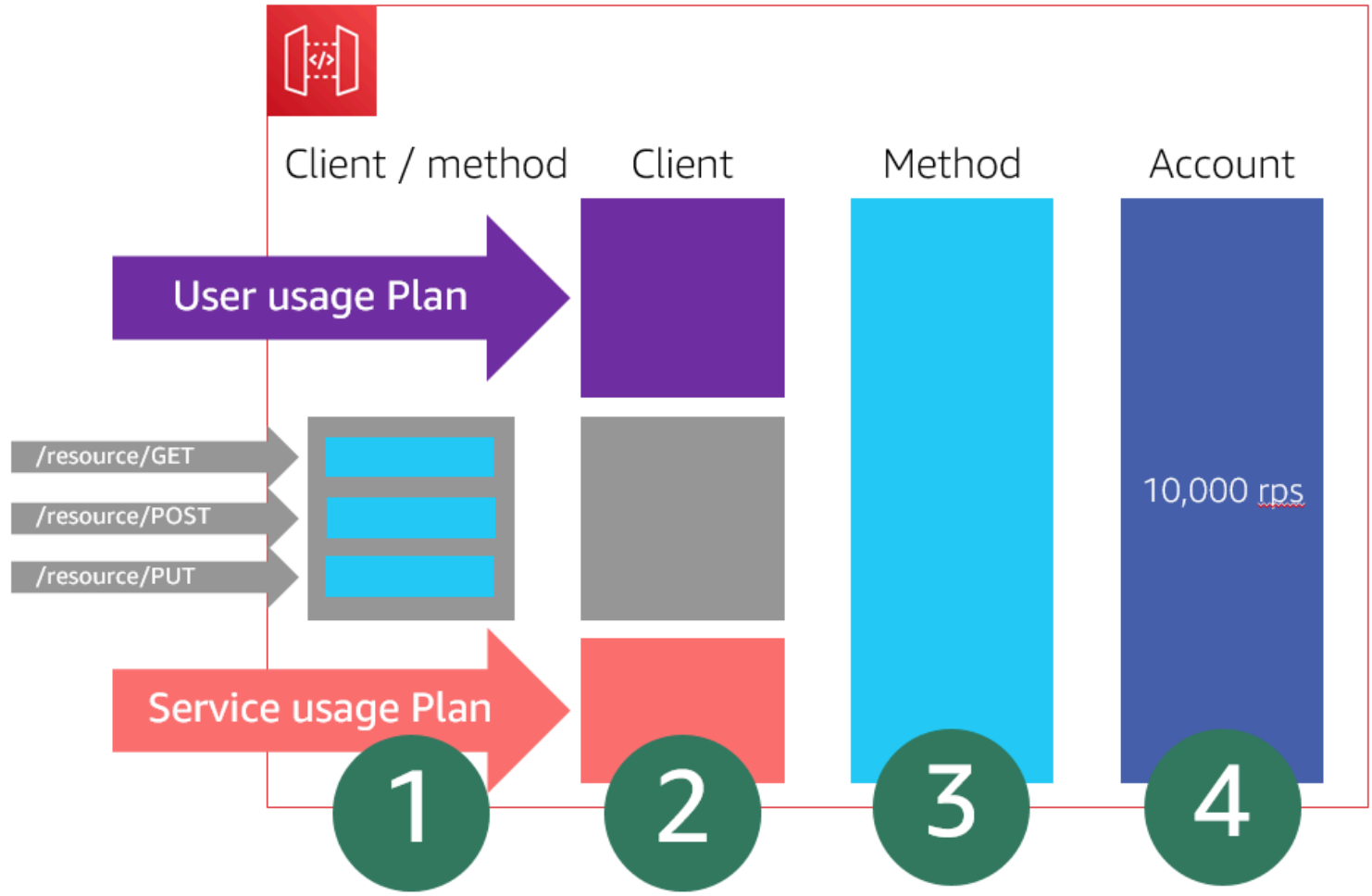
Token bucket algorithm

API Gateway limits the steady-state *rate* and *burst* requests per second. These are shared across all APIs per Region in an account. For further information on account-level throttling per Region, see the documentation. You can request account-level rate limit increases using the AWS Support Center. For more information, see Amazon API Gateway quotas and important notes.

You can configure your own throttling levels, within the account and Region limits to improve overall performance across all APIs in your account. This restricts the overall request submissions so that they don't exceed the account-level throttling limits.

You can also configure per-client throttling limits. Usage plans restrict client request submissions to within specified request rates and quotas. These are applied to clients using API keys that are associated with your usage policy as a client identifier. You can add throttling levels per API route, stage, or method that are applied in a specific order.

For more information on API Gateway throttling, see the AWS re:Invent presentation "I didn't know Amazon API Gateway could do that".



API Gateway throttling

You can also throttle requests by introducing a buffering layer using Amazon Kinesis Data Stream or Amazon SQS. Kinesis can limit the number of requests at the shard level while SQS can limit at the consumer level. For more information on using SQS as a buffer with Amazon Simple Notification Service (SNS), read "How To: Use SNS and SQS to Distribute and Throttle Events".

Identify steady-rate and burst rate requests that your workload can sustain at any point in time before performance degraded

Load testing your serverless application allows you to monitor the performance of an application before it is deployed to production. Serverless applications can be simpler to load test, thanks to the automatic scaling built into many of the services. During a load test, you can identify quotas that may act as a limiting factor for the traffic you expect and take action.

Perform load testing for a sustained period of time. Gradually increase the traffic to your API to determine your steady-state rate of requests. Also use a burst strategy with no ramp up to determine the burst rates that your workload can serve without errors or performance degradation. There are a number of AWS Marketplace and AWS Partner Network (APN) solutions available for performance testing, Gatling Frontline, BlazeMeter, and Apica.

In the serverless airline example used in this series, you can run a performance test suite using Gatling, an open source tool.

To deploy the test suite, follow the instructions in the GitHub repository perf-tests directory. Uncomment the `deploy.perftest` line in the repository Makefile.

```
deploy: ##=> Deploy services
	$(info [*] Deploying...)
	$(MAKE) deploy.payment
	$(MAKE) deploy.booking
	$(MAKE) deploy.loyalty
	$(MAKE) deploy.log-processing
## Enable the deploy.perftest if you need to deploy the performance test stack
#	$(MAKE) deploy.perftest
```

Perf-test makefile

Once the file is pushed to GitHub, AWS Amplify Console rebuilds the application, and deploys an AWS CloudFormation stack. You can run the load tests locally, or use an AWS Step Functions state machine to run the setup and Gatling load test simulation.

Performance test using Step Functions

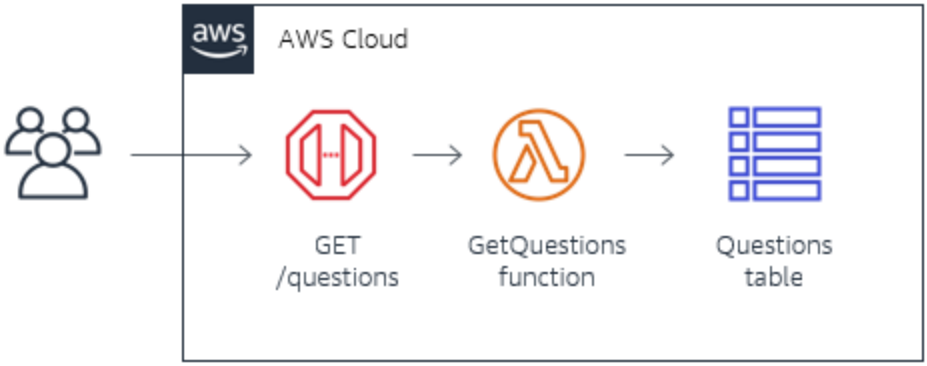
The Gatling simulation script uses `constantUsersPerSec` and `rampUsersPerSec` to add users for a number of test scenarios. You can use the test to simulate load on the application. Once the tests run, it generates a downloadable report.

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	793	793	0	0%	12.79	30	113	326	840	4118	5477	303	532
Search Rand Flights	128	128	0	0%	2.065	35	68	85	115	193	220	73	30
Search Flights	118	118	0	0%	1.903	34	66	87	162	426	451	85	73
Stripe tokenization	128	128	0	0%	2.065	295	322	340	444	532	650	339	52
Charge API	128	128	0	0%	2.065	677	782	860	4281	4864	5477	1073	979
Retrieve Profile	86	86	0	0%	1.387	94	193	224	467	1293	1324	234	185
List bookings	77	77	0	0%	1.242	30	80	96	129	249	251	84	37
Process booking	128	128	0	0%	2.065	59	98	113	144	322	423	104	44

Gatling performance results

Artillery Community Edition is another 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. 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, which runs the Artillery package on Lambda functions. As a result, this tool can scale up to a significantly higher number of tests.

For more information on how to use Artillery, see "Load testing a web application's serverless backend". This runs tests against APIs in a demo application. For example, one of the tests fetches 50,000 questions per hour. This calls an API Gateway endpoint and tests whether the AWS Lambda function, which queries an Amazon DynamoDB table, can handle the load.



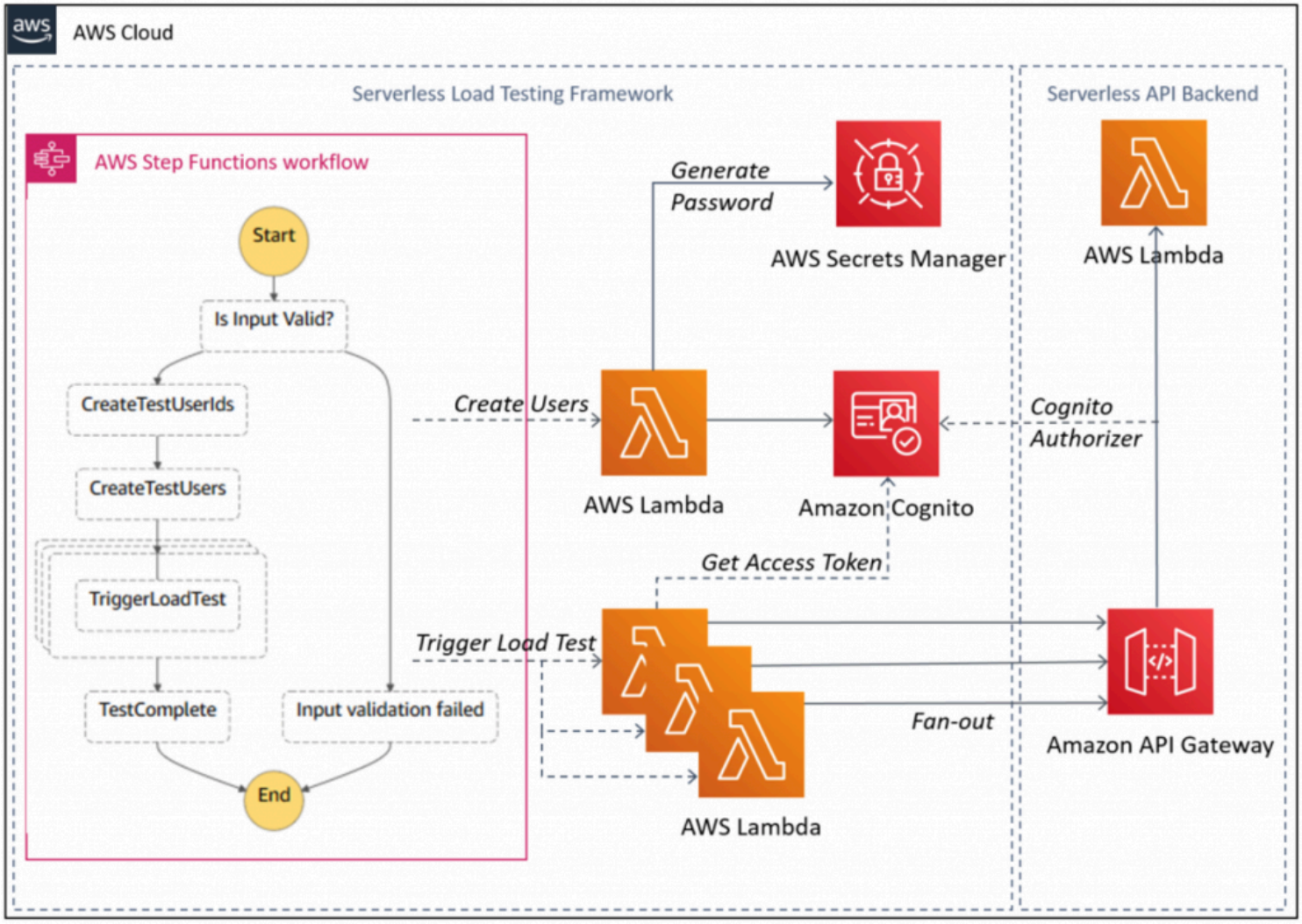
Artillery performance test

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.

```
All virtual users finished
Summary report @ 13:43:46(+0000) 2020-05-14
  Scenarios launched:  2400
  Scenarios completed: 2400
  Requests completed:  2400
  Mean response/sec: 19.93
  Response time (msec):
    min: 131.7
    max: 406.7
    median: 165.4
    p95: 200.9
    p99: 233
  Scenario counts:
    0: 2400 (100%)
  Codes:
    200: 2400
```

Performance test API results

Another consideration for API load testing is whether the authentication and authorization service can handle the load. For more information on load testing Amazon Cognito and API Gateway using Step Functions, see "Using serverless to load test Amazon API Gateway with authorization".



API load testing with authentication and authorization

Conclusion

Regulating inbound requests helps you adapt different scaling mechanisms based on customer demand. You can achieve better throughput for your workloads and make them more reliable by controlling requests to a rate that your workload can support.

In this post, I cover controlling inbound request rates using throttling. I show how to use throttling to control steady-rate and burst rate requests. I show some solutions for performance testing to identify the request rates that your workload can sustain before performance degradation.

This well-architected question continues in part 2 where I look at using, analyzing, and enforcing API quotas. I cover mechanisms to protect non-scalable resources.

For more serverless learning resources, visit Serverless Land.
TAGS: serverless, well-architected