Taking Varnish API Engine for a test drive

We took the Varnish API Engine out for a spin under a few different conditions, and here is what we found.

Scope

The scope of this test is to indicate the performance impact of introducing the Varnish API Engine, a high performance API gateway, in front of very fast web servers in various scenarios using synthetic load. The synthetic load is carefully selected to match various scenarios to trigger typical API gateway features.

The tests are deliberately done without caching. This to put focus on the performance of the authentication, authorization and throttling features of the API Engine. In a real-world scenario performance can be expected to be significantly higher, as at least some part of your APIs can most likely be cached.

The Varnish API Engine environment in this test consists of a three-node cluster and management nodes. The base subscription of the API Engine includes three servers running API Engine to process API calls.

Methodology

Initially a *reference test* is performed without the Varnish API Engine. Then a set of tests is performed with Varnish API Engine, and the results from these tests are compared to the reference test to identify the change in response and performance characteristics.

Components used

- Boom [https://github.com/rakyll/boom] is used to generate load. It is written in Go, and its feature set is similar to Apache Bench. Two patches have been added to enable connections to be reused and to support custom host header [https://github.com/espebra/boom].
- Dummy API [https://github.com/espebra/dummy-api] is used as the web server. This is a simple and easy-to-install web server written in Go using net/http. It should be noted that the Dummy API is unrealistically fast. Any real world API will most likely be orders of magnitude slower. In addition to being run within a highly performant runtime the Dummy API has no external data sources, such as databases or other network based APIs behind it.

- Varnish API Engine 2.0 with Varnish Cache Plus 4 is used as the API gateway.

Reference test

As a reference, we measure the performance of the web servers directly as seen from the consumer instances. The test is done by sending GET requests to a specific resource on the web server, to which the web server responds with the http status code 200 and a small response body.

The performance characteristics that are measured are the requests per second, the slowest transaction, the fastest transaction, the average transaction time and the 99 percentile transaction time.

Test scenarios

The reference test is compared to the following scenarios where the mentioned tasks are introduced and handled by Varnish API Engine:

Test scenario 1: Authentication and authorization. The requests are accepted with http status code 200 sent from the backend.

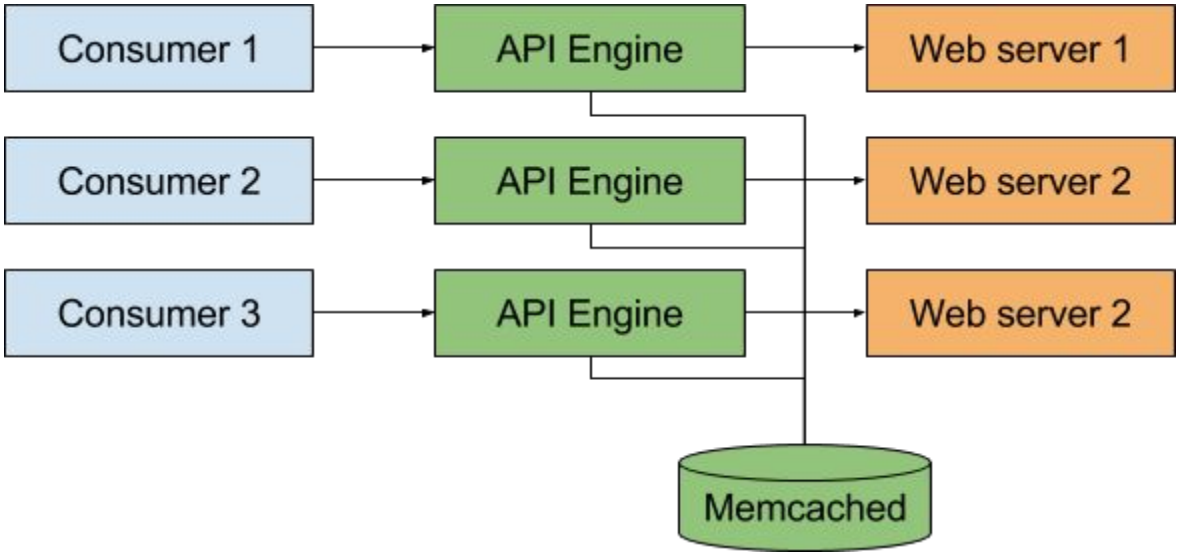
Test scenario 2: Authentication, authorization and one rate limit. The requests are accepted with http status code 200 sent from the backend.

Test scenario 3: Authentication, authorization and two rate limits. The requests are accepted with http status code 200 sent from the backend.

Test scenario 4: Authentication, authorization, two rate limits and HTTP method filter. The requests are accepted with http status code 200 sent from the backend.

Test scenario 5: Authentication, authorization and one rate limit which allows only one request. The rest of the requests are throttled, meaning rejected with http status code 429 sent from the Varnish API Engine.

Test scenario 6: Authentication where the consumer tries to authenticate using an unknown api key. The requests are rejected with http status code 401 sent from the Varnish API Engine.



Execution

Boom, which is used by the three consumer instances to generate load, is executed with the following arguments in all tests:

```
-H "api.example.com" -n 100000 -c 100
```

The argument `-H` sets the custom host header, `-n` sets the amount of requests, and `-c` sets the concurrency.

Boom is executed simultaneously on each of the three consumers. The consumers generate load towards each of the three Varnish API Engine instances (test scenarios) or the three web servers (reference test).

Example command from the reference test:

```
consumer1$ boom -H "api.example.com" -n 100000 -c 100
http://backend1:1337/test/?max-age=0&apikey=t1
```

Example command from one of the test scenarios:

```
consumer3 $ boom -H "api.example.com" -n 100000 -c 100
http://varnish3:6081/test/?max-age=0&apikey=t3
```

Uncertainties

Standard deviations and averages over multiple test runs are not calculated. The performance characteristics may vary between cloud instances of the same type.

Test setup

Environment

The environment consists of 12 instances running in Amazon EC2 within the same availability zone (us-east-1c) and placement group. The instances are running the CentOS 7 x86_64 EBS HVM AMI on m4.xlarge instances with 4 vCPUs, 16 GiB memory, 8 GB SSD general purpose storage and *Network Performance* rated as high.

The instances are used as follows:

- 1 instance running the Varnish API Engine management
- 1 instance running Varnish Custom Statistics
- 1 instance running Memcached for accounting
- 3 instances running Varnish Cache Plus with Varnish API Engine
- 3 instances running a web server acting as backends
- 3 instances acting as consumers to generate load

The consumers communicate with the three Varnish instances directly, without using an Elastic Load Balancer (ELB). The consumers put load simultaneously on the the web servers (reference test) and the Varnish instances (test scenarios). In a real world scenario the ELB would most likely be used, but as it limits performance in stress tests we bypassed it. Using a more realistic, i.e. a slower API would make it harder to showcase the performance of the API Engine.

In most real world scenarios the API Engine will not inhibit API performance.

Dummy API is running on the web servers instance and binds to host 0.0.0.0 and port 1337.

Installation and configuration

Ansible <http://www.ansible.com/> is used to install and configure the environment. The ansible configuration is available at Github <https://github.com/varnish/api-engine-performance-tests>.

Test scenarios and results

Reference test

Three consumers generate load against one backend each. The result from each consumer is then summarized to get an aggregated result for the three consumers, which is the reference result.

Requests per second	41706
Successful transactions	300000
Failed transactions	0
Slowest transaction time	0.1525 seconds
Fastest transaction time	0.0003 seconds
Average transaction time	0.0071 seconds
99% of the transactions completed within	0.0189 seconds

The reference results are the aggregated values over the three consumer hosts. As you can see the number of transactions per second is around 42K. There is some jitter in the transaction flow but this should be expected on a virtualized platform.

Test scenario 1 (Authentication and authorization)

Three consumers generate load against one Varnish instance each. The requests are authenticated, authorized and let through to the backends. The backend responses are then delivered to the consumers. 200 responses are considered as successful transactions.

Requests per second	30334
Successful transactions	300000

Failed transactions	0
Slowest transaction time	0.2532 seconds
Fastest transaction time	0.0008 seconds
Average transaction time	0.0099 seconds
99% of the transactions completed within	0.0361 seconds

Here we are down to around 30K. The overhead of proxying the request through another HTTP layer reduces the throughput as expected.

Test scenario 2 (Authentication, authorization and one rate limit)

Three consumers generate load against one Varnish instance each. The requests are authenticated, authorized, verified against one rate limit, and let through to the backends. The backend responses are then delivered to the consumers. 200 responses are considered as successful transactions.

Requests per second	25976
Successful transactions	300000
Failed transactions	0
Slowest transaction time	0.2421 seconds
Fastest transaction time	0.0010 seconds
Average transaction time	0.0115 seconds
99% of the transactions completed within	0.0366 seconds

As you can tell the throughput is reduced further. This is because rate-limiting requires the API Engine to utilize a network service for accounting purposes. This is done synchronously for accuracy, and will add an unavoidable slowdown to the stress test. However, performance is still well within what we're aiming for.

Test scenario 3 (Authentication, authorization and two rate limits)

Three consumers generate load against one Varnish instance each. The requests are authenticated, authorized, verified against two rate limits, and let through to the backends. The

backend responses are then delivered to the consumers. 200 responses are considered as successful transactions.

Requests per second	23623
Successful transactions	300000
Failed transactions	0
Slowest transaction time	0.1560 seconds
Fastest transaction time	0.0012 seconds
Average transaction time	0.0126 seconds
99% of the transactions completed within	0.0292 seconds

As you can see another rate-limit has reduced the throughput further. The request rate is still very much acceptable.

Test scenario 4 (Authentication, authorization, two rate limits and HTTP method filter)

Three consumers generate load against one Varnish instance each. The requests are authenticated, authorized, verified against two rate limits, an HTTP method filter and let through to the backends. The backend responses are then delivered to the consumers. 200 responses are considered as successful transactions.

Requests per second	23164
Successful transactions	300000
Failed transactions	0
Slowest transaction time	0.1912 seconds
Fastest transaction time	0.0013 seconds
Average transaction time	0.0129 seconds
99% of the transactions completed within	0.0276 seconds

As you can see the overhead of the HTTP method filtering has added a marginal slowdown. This is basically a worst case scenario where all the features have been enabled.

Test scenario 5 (Authentication, authorization, strict rate limiting)

Three consumers generate load against one Varnish instance each. The requests are authenticated, authorized, verified against one rate limit that will exceed immediately. The consumers are then rejected with 429 Too Many Requests responses. The rate limit is configured to allow 1 request per 30 seconds. The initial request is allowed, and the rest of the requests are rejected. One 200 response and the rest being 429 responses are considered as successful transactions.

Requests per second	45241
Successful transactions	300000
Failed transactions	0
Slowest transaction time	0.8096 seconds
Fastest transaction time	0.0006 seconds
Average transaction time	0.0063 seconds
99% of the transactions completed within	0.2016 seconds

As the requests here get rate-limited and are rejected the work the API Engine has to do on each requests is reduced dramatically. The major contributor here is most likely due to the API Engine not having to actually talk to the underlying API, but it is able to reject most of the requests outright.

Test scenario 6 (Authentication failures)

Three consumers generate load against one Varnish instance each. The requests fail the authentication and authorization check, and are rejected immediately. The requests are rejected with 401 Unauthorized. 401 responses are considered as successful transactions.

Requests per second	73339
Successful transactions	300000
Failed transactions	0
Slowest transaction time	0.1547 seconds
Fastest transaction time	0.0003 seconds

Average transaction time	0.0041 seconds
99% of the transactions completed within	0.0193 seconds

Here, as the incoming requests fail to pass the authentication and authorization checks the API Engine can stop processing the request without even consulting the accounting service. A minimal amount of processing power is consumed. This is more or less what you would expect a bare bones Varnish Cache installation to perform with ~100% hit rate.

Scalability

One crucial aspect of the design is scalability. Some of the prospects currently testing the API Engine have requirements north of 100K API calls per second. Servicing these kind of volumes means we both have to have a performance per server that is adequate to reach these numbers with a reasonable amount of servers and that we can scale horizontally without any particular component becoming a bottleneck.

In addition to the API Engine, which runs within Varnish Cache, the other component that needs to scale is memcached, which is used for accounting. As the number of requests grows we can add more memcache servers. API Engine to memcache ratio seems to be about 1:4 or so, depending on usage.

Interpreting the results

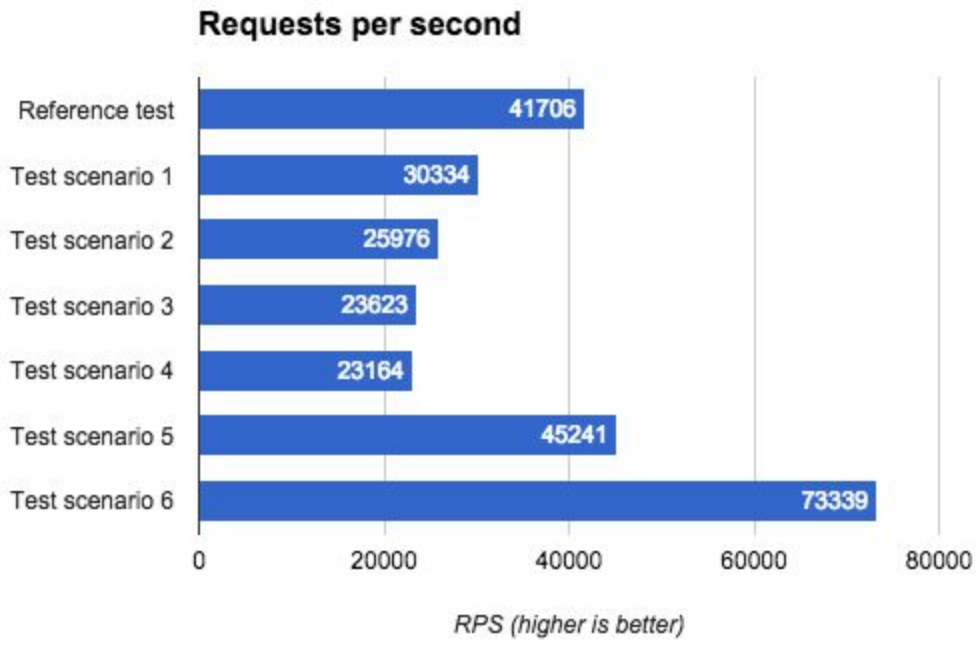
So what do we actually see here? How are we to interpret the stress test? As with every benchmark we have to take it all with a grain of salt. In a real-world scenario there are conditions that are hard to replicate in a limited stress test. Factors such as packet loss and latencies will most likely affect real-world capacity. As with any scalability challenge, one should be conservative when estimating performance as the cost of getting it wrong is usually quite high.

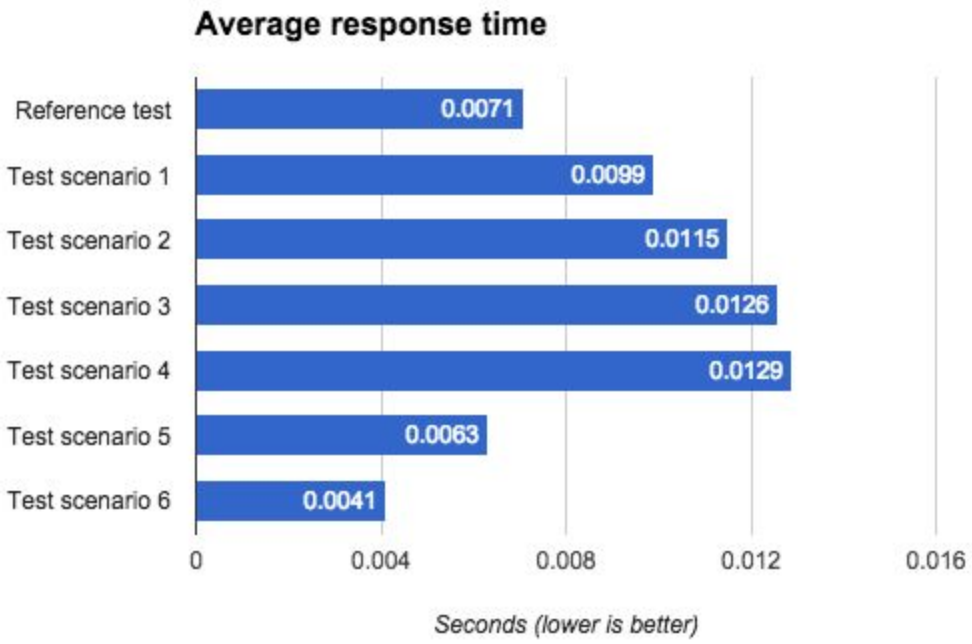
The tests we've done give us a baseline capacity of processing around 23K API calls per second on a three-node API Engine cluster. Mobile clients will drag it down a bit. This performance hit will most likely be more than mitigated by Varnish API Engine's ability to cache some of the responses, and reject throttled and unauthorized requests. We believe that 23K per second is very much achievable in a real-world scenario.

As for latency, any HTTP proxying layer will add a bit of latency. Varnish itself seems to add around 2.7 ms to each request. When we start doing throttling we increase the latency a bit, as the API Engine talks to memcached in order to keep track of the number of requests flowing through the cluster.

All in all, we're very happy with the performance.

Test	Description
Reference	Direct API access.
Scenario 1	Authentication and authorization, all requests accepted.
Scenario 2	Authentication, authorization and one rate limit, all requests accepted.
Scenario 3	Authentication, authorization and two rate limits, all requests accepted.
Scenario 4	Authentication, authorization, two rate limits and HTTP method filter, all requests accepted.
Scenario 5	Authentication, authorization and one rate limit, only one request accepted.
Scenario 6	Authentication, all requests denied.




Average response time
Reference test
Test scenario 1
Test scenario 2
Test scenario 3
Test scenario 4
Test scenario 5
Test scenario 6
0.0071
0.0099
0.0115
0.0126
0.0129
0.0063
0.0041
0
0.004
0.008
0.012
0.016
Seconds (lower is better)