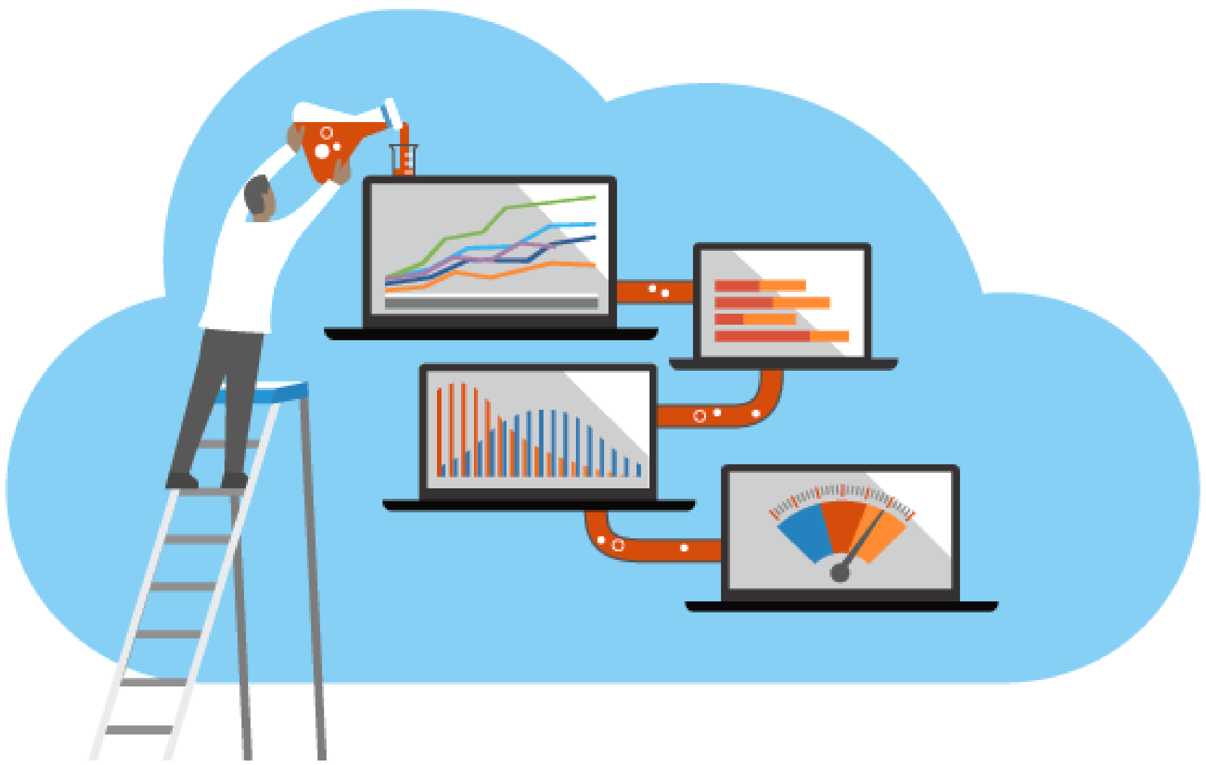
Improving Fastly's HTTP Load Testing Framework



Stefano Jordhani
Ashwin Pai
Saniya Syeda

An Major Qualifying Project
submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfilment of the requirements for the
degree of Bachelor of Science

by
Ashwin Pai
Saniya Syeda
Stefano Jordhani

Date:
B Term 2021

Report Submitted to:

Patrick McManus
Joe Damato
Salman Saghafi
Fastly

Professor Mark Claypool
Worcester Polytechnic Institute

ABSTRACT

Fastly is a cloud computing services provider seeking to improve their load-testing technologies. Currently, they are unable to generate enough network traffic to accurately represent a production environment and as a result, cannot precisely measure server performance. This project aimed to improve the range of tests Fastly can run on their software stack, allowing them to detect and reproduce production issues such as memory allocation for idle connections and throughput when processing TLS handshakes and requests concurrently. To achieve this goal, our team conducted comparative analysis of existing load-testing tools and extended PTR, an internal Fastly project. Some features we integrated include support for multiple parallel tests and generating load from K6, an open-source HTTP testing framework, as well as Idle Connections, a tool we wrote that generates connected yet dormant clients. We ran PTR against Fastly's infrastructure to recreate complex scenarios such as comparing performance across machines with different processors and server builds. From our results, we were able to uncover potential bottlenecks along with other server inefficiencies. By identifying and fixing these issues, Fastly can help provide reliable service to their customers.

ACKNOWLEDGEMENTS

We would like to acknowledge the following individuals for their contribution towards our project. First, we would like to thank our advisor, Professor Mark Claypool for his valuable feedback and advice and being a source of constant support. We would also like to thank our sponsor, Fastly, for welcoming us and making us feel like a part of the team. Finally, we would like to thank Joe Damato and Salman Saghafi, our Fastly mentors, for all the work they did in guiding us throughout this project. We appreciate the positive learning environment that was provided to us. The team is grateful and truly enjoyed working with all of them. This project would not have been nearly as successful without their time and support.

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

CHAPTER 1: Introduction

When companies who produce multi-user systems release new features or advertise campaigns, they may see a significant increase in network traffic and need a way to evaluate site performance beforehand. HTTP load-testing allows developers to analyze how their application supports an expected workload that emulates customer behavior. This can help reveal potential problems before users encounter them.

H2Load is a HTTP load-testing framework used by Fastly. This tool has uncovered issues in the software stack that would have been challenging to reproduce in a production environment. H2Load is compiled with nghttp2 which is an open source implementation of HTTP/2 (Tsujikawa, 2015) and is compatible with HTTP/1.1 and HTTP/2. H2Load supports SSL/TLS and clear text for both protocols (Tsujikawa, 2016).

While H2Load provides many advantages, there are also some limitations that have become apparent as technology and HTTP advance. Two such limitations, for example, include the inability to generate detailed metrics or print output in JSON format. Additionally, on its own, H2Load is unable to generate network traffic to accurately represent a production environment; as a result, Fastly cannot precisely measure server performance. Although H2Load has proved to be helpful, as Fastly's backend begins to accommodate new technologies, they a more comprehensive and flexible load-testing tool.

Our goal is to improve the range of tests that Fastly can run on their software stack and allow them to detect any issues that are presented by the addition of new technologies before their products are released to customers. Fastly was looking for a HTTP load-testing tool that supported the following features:

- HTTP version 1 and 2
- Issuing requests/load over TLS protocol
- Command-line interface
- Gathering and reporting detailed metrics such as time to first byte (TTFB), request completion time, bandwidth metrics, latency, and so on, perhaps categorized by request type and URL pattern

- JSON output

Additional features that would be beneficial include:

- Controlling the TLS handshake rate
- Issuing requests with varying HTTP methods (i.e. GET, POST, PUT, etc…) and target URLs (i.e. example.com/test1, example.com/test2, etc…) in a single test run
- Issuing requests that have different headers
- Generating idle connections against a target
- Currently up-to-date or being regularly updated
- Issuing requests on a combination of protocols (for example issuing both H1 and H2 requests to the server in a single test run)
- Extensibility for adding new HTTP load testing features in the future

First, we examined existing load-testing technologies to determine if there was a tool that meets Fastly's needs. Once we found K6, an open source load-testing tool that met a majority of Fastly's requirements, we implemented it into an existing framework called Performance Test Runner (Performance Test Runner (PTR). Using PTR, we ran load tests on Fastly's servers in order to recreate complex scenarios seen in production environments. With the results gathered from the tests, we were able to identify potential bottlenecks along with other server vulnerabilities.

Chapter 2 provides background and related technologies on HTTP load-testing. Chapter 3 describes our methodology of researching and selecting a tool. Chapter 4 discusses the implementation of our solution into PTR's existing code base and extending K6. Chapter 5 evaluates the capabilities of PTR and describes the results from the experiments ran against Fastly's infrastructure. Chapter 6 concludes by coalescing our results at a high level. Finally, Chapter 7 presents some future recommendations for Fastly's load-testing system.

CHAPTER 2: Background

Section 2.1 explores HTTP and how it is structured. This describes different processes and protocols of HTTP and the security aspect of HTTPS. Section 2.2 introduces HTTP load testing and its uses, discussing the importance of HTTP load testing for application owners, general guidelines for conducting load testing, and how it is measured. Section 2.3 describes H2Load, the current HTTP testing framework used by Fastly, and also discusses the advantages and limitations of using H2Load.

2.1: What is HTTP?

At a high level, HTTP is a client-server protocol in which requests are sent to a server by a single entity, also known as the user-agent. The server is responsible for handling each request and providing a response back to the client (*An overview of HTTP*, 2021). Figure 1, shown below, displays a high-level representation of how the HTTP process works.

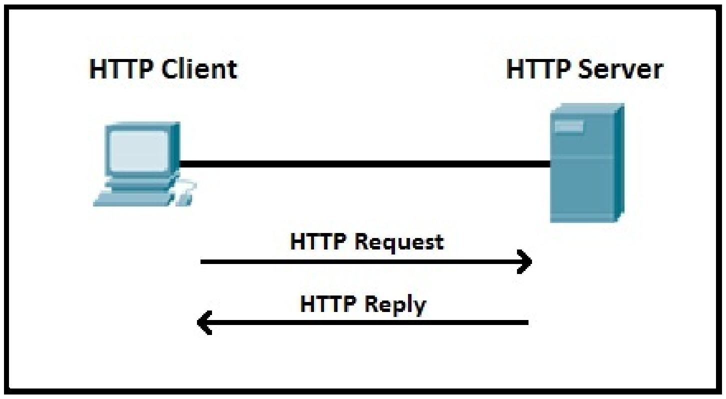


Figure 1: Diagram showing how HTTP works at a high level (*Study CCNA*)

In HTTP, it is the client (or the user-agent) that establishes the connection to the server over which it can send requests (*A typical HTTP session*, 2021). In most situations, the user-agent is a web browser but developers can also implement programs and use tools such as cURL[1] to act as the user-agent (*Hypertext protocol (HTTP): Client-server communication,* n.d.). On the other end

[1] Command-line tool for simulating an HTTP request to server

of the communication channel is the server. Virtually, the server appears as a single entity/machine; however, it can actually be a collection of servers managing the load coming in or a "complex piece of software interrogating other computers" (*An overview of HTTP*, 2021). Overall, an HTTP session consists of three phases seen below (*A typical HTTP session*, 2021):

1.) The client establishes a connection
2.) The client sends a request and waits for an answer
3.) The server handles the request and sends back a response, which includes a status code and appropriate data

HTTP requests and responses have well-defined, specific structures. HTTP requests consist of an HTTP method, the path of the resource to fetch, the version of HTTP protocol in use, optional headers, along with a body that holds the resource to be sent. An HTTP method specifies the action the client wants to take and is usually named after a verb like GET/POST or a noun like OPTIONS/HEAD (*Hypertext protocol (HTTP): Client-server communication,* n.d.). For example, as the name suggests, a GET operation is used to fetch a resource from the server. On the other hand, an HTTP response consists of the version of HTTP protocol in use, a status code (as mentioned earlier), a status message, HTTP headers, and a body that holds the content. The HTTP request and response share some similarities, one of which is an HTTP header.

HTTP headers allow the client and server to provide additional information (metadata) with an HTTP request or response. Headers can be categorized as request headers, response headers, representation headers, or payload headers (*HTTP headers*, 2021). At a high level, request headers hold information regarding the resource to be fetched, or about the client requesting the resource. Response headers hold information such as the location of the response or additional information about the server sending the response. Representation headers hold information about the body being sent along with the HTTP message. Finally, payload headers hold representation-independent information about payload data.

2.1.1: Different versions of HTTP protocols

While HTTP version 1.1 is the most commonly implemented protocol, it has some limitations that have been addressed by newer versions such as HTTP/2 and (Biketi, 2021). When in use, HTTP/1.1 was regarded as having slow response times and as the Internet became more popular, there was a desire to reduce latency[2] to improve user experience by having faster page load times. In addition to decreasing latency, HTTP/1.1 is ineffective in resource prioritization and unable to efficiently manage traffic by compressing HTTP headers. HTTP/2 provides features to improve this area of HTTP communication. One of these features included header compression which reduced the amount of data being sent as it made the transition to use a binary format for HTTP headers. The reduction of information being passed through headers decreased latency thereby increasing overall efficiency. HTTP/2 also introduced the ability to send multiple requests over the same connection[3], allowing requests to be handled in an asynchronous and resource-efficient manner. In addition to that, HTTP/2 allowed for the ability to prioritize certain resources to be sent first from the client/server and introduced the idea of having the server sending resources before they are even requested by the client (Grigorik, Surma, n.d.). All of these features improved the process in which information is shared across an HTTP connection.

2.1.2: Security through HTTPS and TLS

HTTPS and TLS (which has now replaced SSL) deal with the security of communication between client and server. HTTPS is the secure version of HTTP and websites that use SSL/TLS certificates signal to the users that the domain is secure and allows for protected transactions (Gudeliauskas, 2021). The motivation behind using HTTPS is to encrypt data being sent over a HTTP connection (*What is HTTPS?*, n.d.). This protocol is crucial in protecting sensitive information such as passwords and credit card numbers when transmitted across the Internet. Figure 2 below depicts what a hacker would see if a user entered their password into a site with and without HTTPS. Without HTTPS, the hacker is able to clearly see the password that was

[2] Latency is known as “the time that it takes for data to pass from one point on a network to another” (*What is latency?*, 2021).

[3] This idea of sending multiple requests over one connection is referred to as “multiplexing.”

entered by the user. However, with HTTPS, the hacker sees the encrypted password which seems like a random set of bytes.

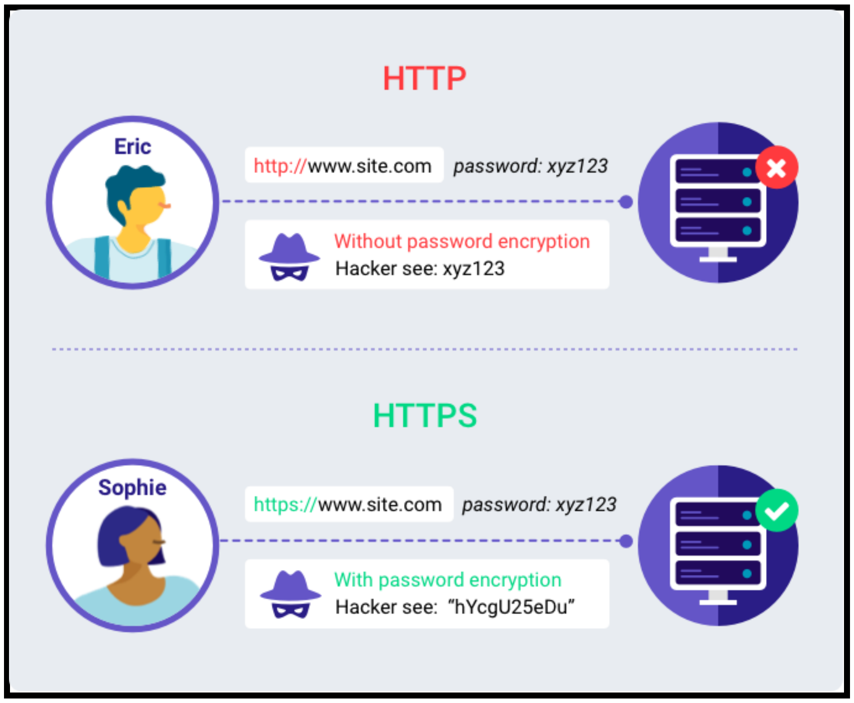


Figure 2: Image showing the differences in what a hacker sees between HTTP and HTTPS (Botify, 2019)

TLS is the protocol that HTTPS uses to allow for encrypted communication (*What is HTTPS?*, n.d.). A communication session that uses TLS is initiated by a TLS handshake that occurs between the server and the client. During a TLS handshake, the client and the server exchange initial messages, verify one another, select an encryption algorithm, and lastly exchange keys to use for encrypting and decrypting data throughout the session (*What happens in a TLS handshake?*, n.d.). The exact steps of the TLS handshake vary based on the encryption algorithm that is used, however, the most common is the RSA key exchange algorithm. The first step of the TLS handshake begins when the client initiates the process by sending a "hello" message which includes the TLS version, cipher suites that the client supports, and a string of random bytes known as the client random. In response to the client's hello message, the server sends the "server hello message" that contains the server's SSL/TLS certificate, the cipher suite used by the server, and the server random, which is similar to the client random. Upon receiving this, the client authenticates the server by verifying the server's TLS/SSL certificate with the authority that issued the certificate. This process ensures that the client is communicating with the owner of the domain. If the client successfully authenticates the server, it sends another random string of encrypted bytes (known as “premaster secret”) that has been encrypted with a public key retrieved from the certificate. To decrypt the random string of bytes, the server uses the private key. If the server successfully decrypts the string, both sides of the communication generate the session keys using the client random, the server random, and the premaster secret. Upon generating the session keys, both the client and the server send a “finished " message to one another using the generated keys. After this final step is completed, the client and server can continue communication through the secure channel. Understanding the structure and processes of HTTP/HTTPS helps in recognizing the importance of the features that Fastly is looking for in a load-testing tool.

2.2: What is HTTP load testing?

HTTP load testing is a practice that many companies use to assess the performance of their services. It primarily involves sending simulated HTTP traffic to a server and measuring how well that server manages under different loads. There are numerous real-world circumstances in the software development process that call for load testing. For example, if a site runs a marketing campaign they often see a significant increase in site traffic; as a result, the site may suffer downtime if the owners are not prepared to accommodate a high volume of users. The same situation also arises when developers are adding new functionality or redesigning pages. Without knowing how new features affect the site load, customers may experience performance problems. With delayed load times, the company loses sales as frustrated users often click away from unresponsive pages. Customer research has shown that 1 in 4 visitors abandon a website that takes more than 4 seconds to load; moreover, 46% of users do not revisit poorly performing websites (Monaghan, 2021). Thus, an insufficiently tested web site runs a higher risk of fewer clients and revenue loss in the future. In fact, about $4.4 billion is lost each year due to poor load performance (Hamilton, 2021). Load testing can help identify and resolve any of these issues before real users have to encounter them.

Another advantage of load testing lies in its ability to assess different parts of an end-to-end IT infrastructure. Developers can isolate and test individual components of the application, such as the front-end, backend, system, client-server communication, or even hardware. Based on the results from the tests, engineers can resolve the issues they discover, improving scalability and efficiency. This process addresses a major challenge that many companies struggle with in knowing where to allocate resources; for example, a business should not try to upgrade their site's servers if customers are only experiencing delay with database retrieval (Menasce, 2002). By dividing up and analyzing each section of the infrastructure, businesses save both time and money.

2.2.1: How HTTP load testing works

Load testing involves some basic guidelines that most developers follow universally. There are a few key questions that should be addressed in the process (Boucheron, 2021):

- Does the server have enough resources - such as CPU and memory - to handle different loads?
- Does the server respond quickly enough to fit usability standards?
- Are there any pages or API calls that are particularly resource intensive?
- How efficiently is the application using resources?

By answering the above questions, developers can extract a number of insights, such as the maximum operating capacity of an application and the number of users it can support. But in order to do so, developers must first determine how to measure performance in a quantifiable way so as to verify success rate. Metrics can fall into one of two categories: response, which reflects the user's perspective, and volume, which represents the traffic generated by the actual load testing tool against the web application (*Load testing metrics explained,* 2017). Both categories provide important insights: response metrics typically tell us what the performance looks like from the outside, while volume metrics explain why, exactly, the application is performing a certain way.

One of the most popular response metrics is “average response time.” As the name implies, this metric is calculated by averaging the total time taken from the start of a client request to the end of the server response (also known as a round trip) over the course of a test cycle. For example, a tester sends a request to view the homepage of a website, and when the full HTML document is built, the server returns the page; the entire time it takes to complete this request-response interaction would be a round trip. The mathematical mean of all round trips in that interval is the average response time. This metric is one of the simplest and most effective in indicating how an application is performing on the user’s end in terms of latency. Some other metrics under the response category include peak response time, which is the longest response time within a test cycle, and error rate, which is the number of errors in comparison to total requests made in the test cycle.

Volume metrics give insight to internal operations in an application that affect performance. One common volume metric is throughput, which is the capacity an application can handle, measured in transactions per second (TPS). If throughput is high, it generally indicates that the application is rapidly transferring a fair amount of response data back and forth. Other volume measurements include concurrent users[4], requests per second, CPU utilization, and memory utilization.

Given these metrics, it is important for developers to follow a certain set of universal guidelines while testing, regardless of differences between applications. A principle rule in general is to test early and test often in the software development lifecycle. This allows the team to identify and correct problems much sooner. Another best practice is to conduct multiple iterative cycles to ensure consistency in results. Generally, the process for these testing life cycles follows a similar set of steps, listed below (Hamilton, 2021):

1. Create a dedicated environment for testing

[4] Concurrent users is the number of active virtual users on the application at a time.

2. Determine or predict load testing transactions and scenarios, for example data needed for each user transaction, number of users accessing the system, connection speeds, and browsers and operating systems used by visitors
3. Execute and test each scenario
4. Collect different metrics and analyze the results
5. Improve the system and iteratively re-test

As previously stated, these testing cycles are designed to follow a feedback loop in which the results from a previous test cycle route back into the next cycle to continuously improve the infrastructure design. This design fits together nicely with the software development lifecycle, especially in Agile or CICD (Continuous Integration Continuous Deployment) environments that emphasize automated and repeated development.

While most follow a basic framework for load-testing practices, companies often differ on the tools they use to actually carry out testing. There are many options available that fit various business needs, each with their own pros and cons. Having a solid understanding of load-testing guidelines is essential in choosing the right tool for Fastly.

2.3 Understanding H2Load as a testing framework

H2Load is an open-source benchmarking tool used for HTTP load testing and is one of the primary frameworks used at Fastly. One of the main advantages of H2Load is that it offers a significant amount of customization when sending HTTP requests. Users can specify a series of flags thereby modifying the operation of a H2Load command and allowing programmers to refine test scenarios. Some of the common flags used in H2Load and their meanings are shown in Table 1 below.

H2Load Flags and Purpose

Flag	Purpose
-n, -n<N>	Number of total requests
-c, --clients=<N>	Number of concurrent clients
-m, --max-concurrent-streams=<N>	Max concurrent streams to issue per session
-H, --Header=<HEADER>	Add/Override a header to the requests
-B, --base-uri=(<URI>|unix:<Path>)	Specify a URI from which the scheme, host and port are used for all requests. (i.e User can specify a UNIX socket as the domain path, as opposed to a TCP socket)
--h1	Forces HTTP1.1 protocol and use of cleartext for TLS and SSL connections.

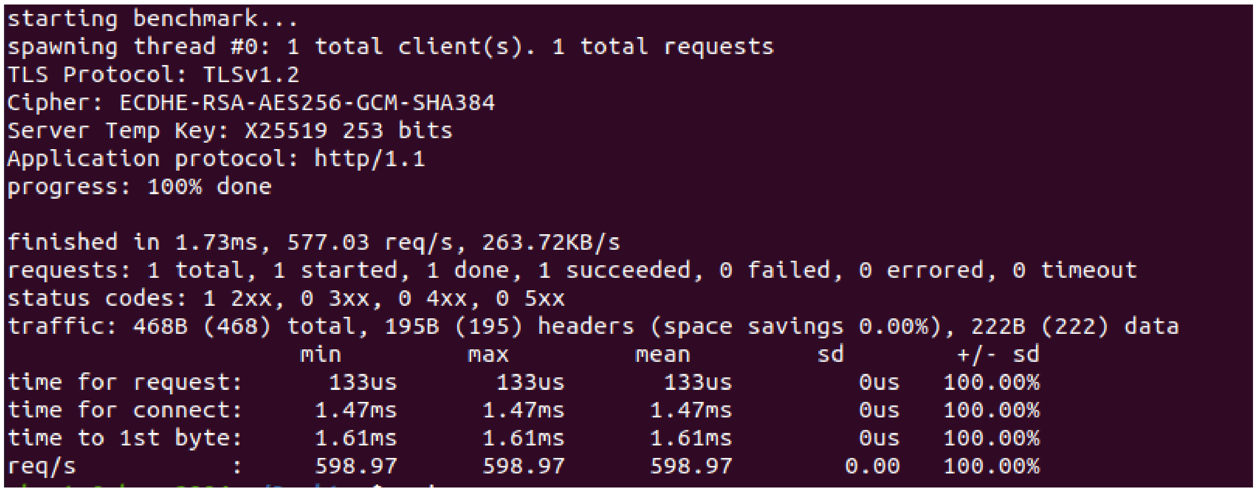
Table 1: Table illustrating the commonly used flags in H2Load requests.

Using a series of flags, programmers can construct HTTP and HTTPS requests to better understand how a server performs under load. Using the table provided in Table 1, it is possible to understand the meaning of the HTTPS request provided in Figure 3 below. Based on the flags provided, the user is sending 1 request, through 1 client using HTTP/1 protocol to a server that is listening on a UNIX “test.sock”, located in the “tmp” directory. The command line options for a H2Load request in conjunction with its related output can help users quickly identify backend issues.

```
H2Load -n1 -c1 -m1 --h1 -B unix:/tmp/test.sock https://localhost:81
```

Figure 3: A sample H2Load request

Another advantage of using the H2Load tool is that the output produced by the requests is helpful in decrypting backend problems and inefficiencies. When sending outgoing requests, H2Load keeps track of relevant information to display to the user once the command has completed its execution. In particular, H2Load measures the total time it takes for a request to be processed, the time it took to connect to the server, the time it took for the server to read the first byte of information, and the number of req/s the client delivered to the server. For each of these statistics, H2Load provides the user with the minimum, maximum, average, and standard deviation. These metrics are useful in understanding if requests are consuming more server bandwidth and resources. In addition, H2Load output also contains information that details the number of requests that succeeded, failed, errored, and timed out along with the associated HTTP status code. Figure 4 below shows the sample output based on the H2Load request shown in Figure 3. In Figure 4, based on the output generated from H2Load, one successful request was sent. In addition, the output shown in the figure indicates that the server spent 133µs processing the request, 1.47ms connecting to the server, and 1.61ms reading the first byte.



```
starting benchmark...
spawning thread #0: 1 total client(s). 1 total requests
TLS Protocol: TLSv1.2
Cipher: ECDHE-RSA-AES256-GCM-SHA384
Server Temp Key: X25519 253 bits
Application protocol: http/1.1
progress: 100% done

finished in 1.73ms, 577.03 req/s, 263.72KB/s
requests: 1 total, 1 started, 1 done, 1 succeeded, 0 failed, 0 errored, 0 timeout
status codes: 1 2xx, 0 3xx, 0 4xx, 0 5xx
traffic: 468B (468) total, 195B (195) headers (space savings 0.00%), 222B (222) data
                     min         max         mean         sd        +/- sd
time for request:      133us       133us       133us         0us   100.00%
time for connect:     1.47ms      1.47ms      1.47ms         0us   100.00%
time to 1st byte:     1.61ms      1.61ms      1.61ms         0us   100.00%
req/s           :     598.97      598.97      598.97        0.00   100.00%
```

Figure 4: The sample output produced by the H2Load command shown in Figure 3

While there are many advantages in using H2Load, it does have a few limitations. H2Load in its current state cannot generate a large amount of TLS handshakes. TLS handshakes are a CPU intensive operation and in production, Fastly has observed that theye have a lot of individual clients establishing TLS handshakes and making requests to their servers. This is seen when there are users or clients clicking around on different webpages running on Fastly's server because there would be, in general, a large number of handshakes and a small number of requests. Considering H2Load does not support this feature, the results that Fastly previously received were inaccurate. With that said, having a load-testing tool that has the capability to control the TLS handshake rate allows Fastly to accurately simulate and observe production traffic.

Another feature that H2Load lacks is the ability to generate idle connections. An idle connection is a connection that is opened on the server that does not issue any requests and remains alive until the server closes it. As an example, this would be the equivalent of a user opening up a webpage but not interacting with it at all, just keeping the page running. The ability to generate idle connections like this would be beneficial to Fastly's load testing needs in particular because of a recent memory consumption issue they were experiencing with h2o, their HTTP server. Specifically, when they modified h2o to increase the default response buffer size from 4kb to 1mb, these 1mb buffers were allocated even for idle connections. This is an issue because idle connections by nature do not issue any HTTP requests, and so millions of clients connecting to a single server instance consumes much more memory than before. With H2Load, there is no way of replicating this scenario and testing it.

Finally, a limitation of H2Load includes the inability to send different requests synchronously with a single command. When sending HTTP requests, H2Load can send a series of requests with the same header to the same endpoint. In certain testing scenarios it may prove useful to construct concurrent HTTP requests with unique headers to measure the performance of the server in this circumstance. As seen in Figure 3, the single H2Load command can only target one server endpoint of "https://localhost:81/." However, in large enterprise companies such as Fastly, there may arise a case in which developers would like to test how a server can handle multiple synchronous requests which are targeting different server-endpoints. In its current implementation, H2Load does not support this task. For the aforementioned reasons, Fastly is in need of a new tool that not only builds upon the advantages seen by H2Load, but also addresses some of its limitations.

CHAPTER 3: Methodology

The following chapter discusses the steps taken to choose a load testing tool that best serves Fastly's needs. Section 3.1 discusses both the mandatory and beneficial requirements for the new load testing tool. Section 3.2 then examines existing load testing tools and objectively ranks them against both sets of criteria. Finally, Section 3.3 concludes with the justification for choosing the load testing tool that best serves Fastly's needs.

3.1: Specifications for the new load-testing tool

In light of the limitations shown by H2Load, Fastly identified a need for a different tool that satisfies a specific set of criteria. Provided below is a set of mandatory features that the load-testing tool must support:

- HTTP version 1 and 2
- Requests and load over TLS protocol
- Command line interface
- Output of detailed metrics such as:
    - Time to first byte
    - Request completion time
    - Bandwidth and latency
- JSON output

Provided below is a set of additional features that would be beneficial but are not mandatory:

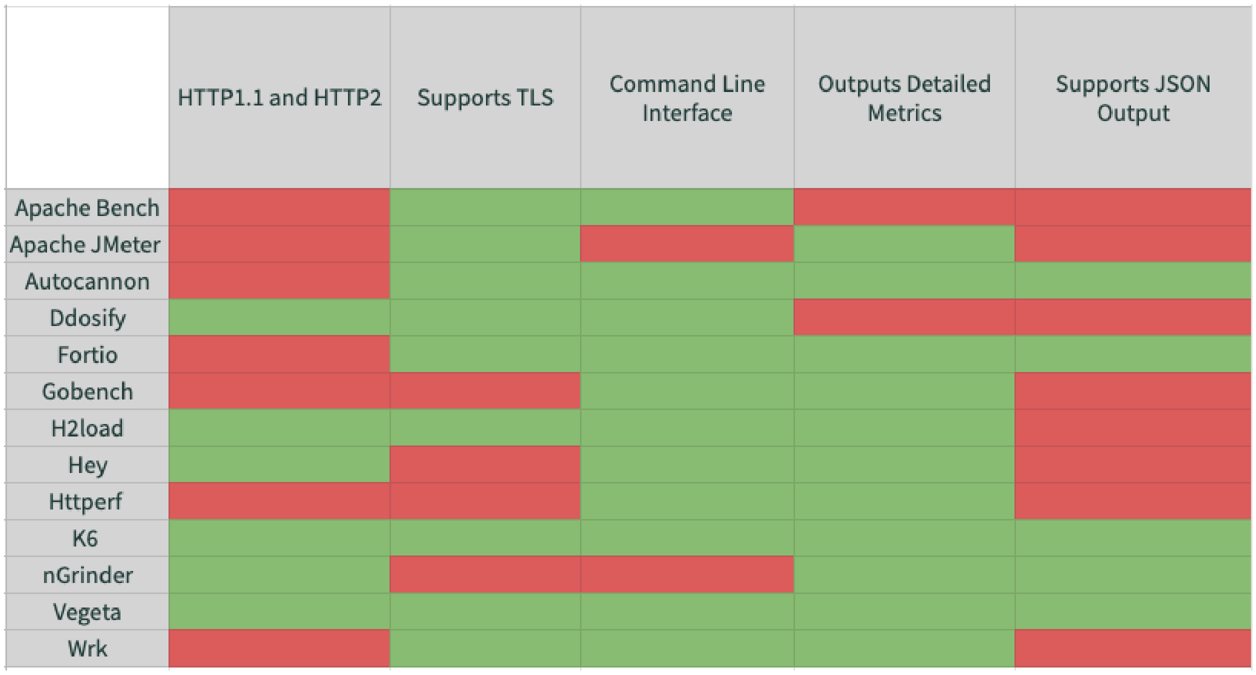
- Controlling the TLS handshake rate
- Issuing requests with varying HTTP methods (i.e. GET, POST, PUT, etc…) and target URLs (i.e. example.com/test1, example.com/test2, etc…) in a single test run
- Issuing requests that have different headers
- Generating idle connections against the target
- Currently up to date or being regularly updated

    ○ Issuing requests on a combination of protocols (for example issuing both H1 and H2 requests to the server in a single test run)
    ○ Extensibility for adding new HTTP load testing features in the future

Given these parameters, we then moved on to researching existing load-testing tools to see if any met Fastly's criteria.

3.2: Researching existing load-testing tools

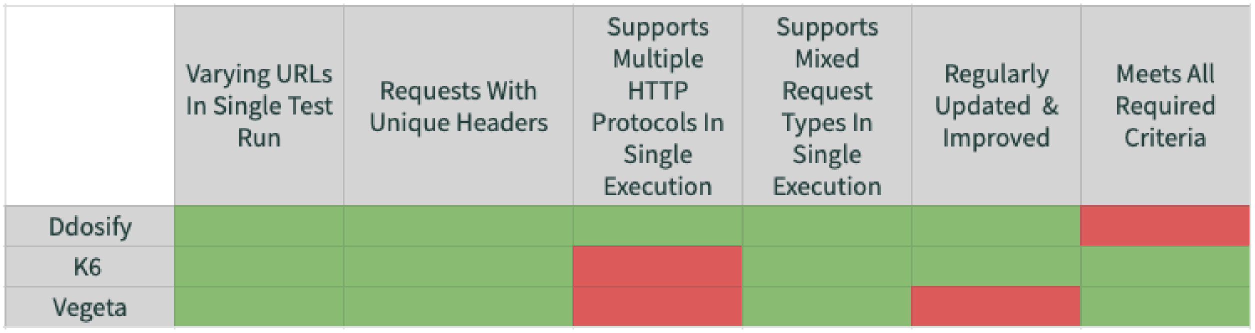
For the preliminary rounds of research for load testing tools, we began by creating a spreadsheet that allowed us to objectively rank each tool against each other. From an initial list of load testing tools, we further investigated the following thirteen: Vegeta, Wrk, H2Load, Apache Bench, Apache JMeter, K6, Autocannon, Gobench, nGrinder, Hey, Httperf, Ddosify, and Fortio. In order to learn about the features provided by each tool, we reviewed the official documentation. Each of these tools were compared to the set of requirements highlighted in Section 3.1. Table 2 showcases the table developed by us when evaluating what features each of the tools supported. Each row represents the name of the tool while the columns highlight each of the mandatory requirements. If a cell is highlighted green, that means the tool fulfills the requirement under that column; if highlighted red, it does not. Column one presents which tools support both HTTP/1.1 and HTTP/2. Since Fastly sends traffic over both HTTP/1.1 and HTTP/2 it is important that the tool can generate traffic using these two HTTP protocols. Column two displays which tools can generate requests over the TLS protocol. Column three shows which tools can be run from the command line - thereby making it a fast and efficient tool runnable from any machine. Column four shows which of the tools display metrics that are helpful to the end user. As shown by the chart, there are some tools that only displayed basic metrics while others tools had more meaningful and in-depth metrics. The final column indicates the tools that can export the resulting metrics to a JSON file. By having metrics stored in a JSON file, it is easier to parse large quantities of data when creating complex load testing scenarios.



	HTTP1.1 and HTTP2	Supports TLS	Command Line Interface	Outputs Detailed Metrics	Supports JSON Output
Apache Bench	🟥	🟩	🟩	🟥	🟥
Apache JMeter	🟥	🟩	🟥	🟩	🟥
Autocannon	🟥	🟩	🟩	🟩	🟩
Ddosify	🟩	🟩	🟩	🟥	🟥
Fortio	🟥	🟩	🟩	🟩	🟩
Gobench	🟥	🟥	🟩	🟩	🟥
H2load	🟩	🟩	🟩	🟩	🟥
Hey	🟩	🟥	🟩	🟩	🟥
Httperf	🟥	🟥	🟩	🟩	🟥
K6	🟩	🟩	🟩	🟩	🟩
nGrinder	🟩	🟥	🟥	🟩	🟩
Vegeta	🟩	🟩	🟩	🟩	🟩
Wrk	🟥	🟩	🟩	🟩	🟥

Table 2: Table of mandatory criteria for initial 13 tools (link to the google sheets)

After our preliminary rounds of research, we narrowed down to three tools: K6, Vegeta, and Ddosify. We eliminated the others based on how many requirements they had met previously. Each team member investigated these tools in-depth to decide which one best met Fastly's requirements. We created a similar table to the one displayed earlier, using the set of "additional," nonessential features for criteria as shown below in Table 3.



	Varying URLs In Single Test Run	Requests With Unique Headers	Supports Multiple HTTP Protocols In Single Execution	Supports Mixed Request Types In Single Execution	Regularly Updated & Improved	Meets All Required Criteria
Ddosify	🟩	🟩	🟩	🟩	🟩	🟥
K6	🟩	🟩	🟥	🟩	🟩	🟩
Vegeta	🟩	🟩	🟥	🟩	🟥	🟩

Table 3: Table of additional criteria for top three tools (link to the google sheets)

"Varying URLs in a single test run" indicates that the program can send load to multiple URLs in a single execution. "Requests with Unique Headers" indicates that the tool can send requests with different headers specified (ex. Host, User-Agent, Accept-Encoding, etc.). "Support Mixed Request Types in Single Execution" means that the tool can send various HTTP methods such as GET and POST to a server in a single execution. "Supports Multiple HTTP Protocols in Single Execution" indicates that the tool can issue requests over different HTTP versions, such as HTTP/1.1 and HTTP/2, in the same run. "Regularly Updated/Improved" means that the tool is currently maintained by its creators, which pertains here since they are all open-source projects on Github. Finally, "Meets All Required Criteria" means that the tool met all the "mandatory" criteria from the previous table in Table 2. One discrepancy is that Ddosify is part of our top three tools, yet did not meet all the "required" criteria; we reasoned that ddosify was worth considering because it fulfilled all of our additional requirements, including one that neither K6 nor Vegeta fulfilled as can be seen in Table 3.

After filling out this sheet, we individually explored the tools' capabilities, installing the tools on our local machines, creating configuration files where applicable, and writing small bash scripts to test them out. As a team, we met and presented demos to illustrate each tool's features and discuss the pros and cons. We compiled our own notes and those from the discussion into a final comparison chart shown in Table 4.

	Pros	Cons
Vegeta	❖ Has sufficient documentation ❖ Uses a target file with list of urls to hit, can customize protocols and headers here ❖ Supports various command line flag options	❖ Has not been updated in over a year ❖ Target file does not allow for any options to be specified
K6	❖ Runs with JavaScript configuration files making it easy to reuse code ❖ Has extensive documentation ❖ Customizable metrics ❖ Able to specify and modify the number virtual users throughout a test scenario ❖ Supports ability to set thresholds for specific metrics ❖ Supports test cases to check status, protocol, HTTP version, etc. ❖ Supports scenario options being specified in a config file	❖ Does not support forcing requests to be sent over HTTP/1.1 protocol ❖ Will send requests over highest protocol supported by the endpoint
Ddosify	❖ Has various forums for feature requests, questions, issues, etc. ❖ Allows for a customized config file in json with multiple "steps" ❖ Supports multiple types of load patterns	❖ Very new tool - lack of documentation ❖ Metrics are not customizable ❖ Less flexibility with writing scenarios because config is written in json

Table 4: Pros and cons table for Vegeta, K6, and Ddosify

Based on the list of pros and cons, as well as our comprehensive evaluation of the tools overall, we determined that K6 was the best tool for Fastly, as it offered marginally greater benefits in terms of customization, documentation, features, and flexibility. One of K6's biggest advantages and one of our deciding factors was its JavaScript configuration file, which allows the user to specify options to run the program with. In the following chapters, we further explore how we utilized K6's features and extended upon them to fit Fastly's needs.

CHAPTER 4: Implementation

The following chapter discusses the steps taken to implement K6 and other tools into a wrapper program. Section 4.1 opens by highlighting one of the limitations of K6 and the steps taken by the team to address this issue. Section 4.2 then transitions into a detailed discussion on the implementation of K6 into an existing Fastly project, known as Performance Test Runner (PTR). This section introduces implementation details of PTR and other regions of the code that help cover K6 limitations. Finally, section 4.3 describes how PTR allows for multiple load-testing tools to be run together.

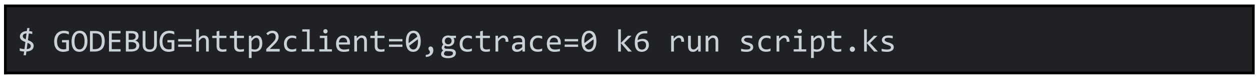
4.1: Extending K6 to support sending requests over specific protocols

Although K6 supports requests over HTTP/1.1 and HTTP/2.0, it does not allow the user to specify which protocol to use. Rather, K6 defaults to the highest protocol supported by the endpoint. For example, if K6 detects that a server supports both H1 and H2, it natively issues requests over HTTP/2.0 protocol. On the other hand, Fastly's previous load-testing tool, H2Load, allowed the user to specify which HTTP protocol to issue requests over. Considering a significant portion of Fastly's traffic is sent over HTTP/1.1 and their previous tool supported this feature, Fastly did not want to lose their ability to test HTTP/1.1 with the new tool.

In order to satisfy this requirement Fastly had for the new load-testing tool, we extended K6 to support a "force HTTP/1.1" feature. Our original pull request against the official K6 project included an option that allowed a K6 user to globally specify the protocol to issue all the requests over. This feature would give the user the ability to set this option inside of a K6 JavaScript configuration file as a command-line flag or as an environment variable. To ensure that our feature followed the existing design as K6, we looked through the codebase to analyze how other global options were implemented before writing it ourselves. Although the global force HTTP/1.1 option was functional, the pull request was rejected because the design of the feature did not align with the future of the K6 project. Instead, the K6 maintainers were working on an improved HTTP API implementation that would allow for a similar feature - making the new global option temporary. This was the primary motive to avoid adding a new global option.

Removing the global option in the future would have resulted in major breaking changes. For instance, if a user began using the force HTTP/1.1 option in their JavaScript configuration files, when it came time to replace the global option with the new HTTP API implementation, it would make those configuration files nonfunctional. With this, the K6 maintainers suggested we leverage the GODEBUG variable to allow the user to only set the force HTTP/1.1 flag on the command line.

The GODEBUG variable is a part of the Go runtime package and is used to control debugging variables during runtime. This variable is a comma-separated list of “name=value” pairs used to set other specific predefined variables. One of these predefined variables is “http2client,” and when it is set to zero as a part of GODEBUG, it disables HTTP/2 client support. At the time of extending K6, the creators had previously removed support for GODEBUG. In order to bring back support, we implemented code to parse the GODEBUG variable inside of K6, and if it found that “http2client=0” was set inside of GODEBUG, it would change the protocol to HTTP/1.1. Figure 5 below shows how the user would run K6 such that all the requests are issued over HTTP/1.1.



```
$ GODEBUG=http2client=0,gctrace=0 k6 run script.ks
```

Figure 5: Screenshot of command showing how to run K6 with the force HTTP1 feature

Our team also added unit and functional tests to the updated pull request to display that the feature was behaving as intended.

4.2: Extending PTR to support other load-testing tools

In order to effectively utilize a tool like K6 for load testing in a production environment, we needed a way to wrap K6. The intention of a wrapper program is to abstract how load-testing tools are run (to the user). In our case, a wrapper program also involves SSHing into multiple virtual machines to generate enough load toward a common endpoint and use multiple tools together. By SSHing into multiple generator machines and targeting a common Fastly production server, we are able to produce enough load to notice a significant impact on server resource consumption. Additionally, with the ability to use multiple tools together, we are able to address K6’s limitations by leveraging other tools and programs.

As part of our research process, we encountered an internal Fastly project called Performance Test Runner (PTR). Originally, PTR was a wrapper program written in Go that wrapped two load-testing tools called H2Load and WRK. Prior to PTR, Fastly used a wrapper program named VLT. While VLT was able to SSH and generate load from multiple machines, it only wrapped H2Load and was not extensible. Considering PTR supported all of the features we were looking for in a wrapper program, we used PTR as an initial framework and extended it to support other tools like K6. A limitation we encountered was PTR’s lack of documentation since the program was not widely used at Fastly. In order to build and deploy this tool, our team looked through the code base to understand the design of PTR. By learning the structure of the project, we were able to integrate our features in a way that conformed to the existing design. In addition to developing features, we also added documentation to help future users at Fastly understand how to leverage PTR for their load-testing needs.

At a high level, PTR is in charge of generating commands for the scenarios that the user specifies in their configuration file, SSH’ing into generator machines and executing the command, and then gathering the output metrics from a tool to report back to the user. Figure 6 below is a diagram representing the steps PTR executes from the beginning to the end of execution.

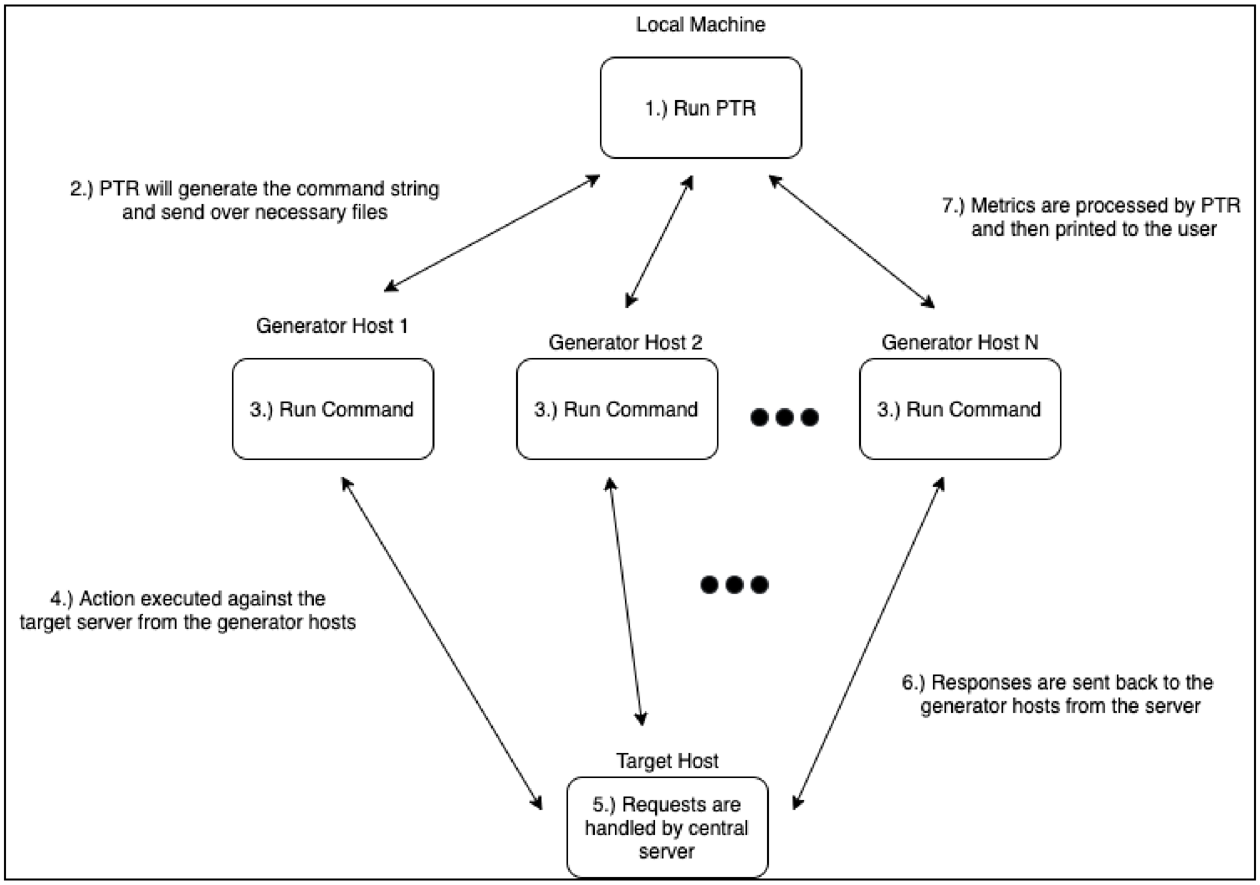

Local Machine
1.) Run PTR
2.) PTR will generate the command string and send over necessary files
7.) Metrics are processed by PTR and then printed to the user
Generator Host 1
Generator Host 2
Generator Host N
3.) Run Command
3.) Run Command
3.) Run Command
4.) Action executed against the target server from the generator hosts
6.) Responses are sent back to the generator hosts from the server
Target Host
5.) Requests are handled by central server

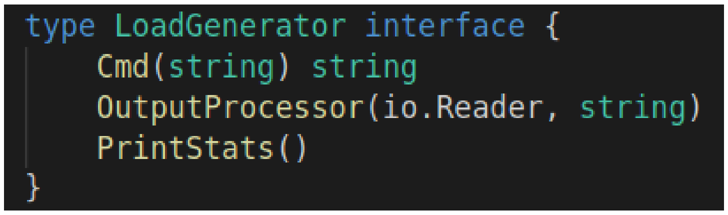
Figure 6: Diagram depicting the steps that are taken during PTR's execution

As shown in the diagram, step one in the process involves running PTR on a user's local machine. Next, after parsing the configuration file, PTR generates the appropriate command string for the tool being run and creates threads to SSH into each of the specified generator hosts. Step three involves the same command being run on each generator host, in which an action is executed against a central target host. In the case of running PTR with K6, step four includes each generator host sending identical requests to the target. Once the generator hosts execute a command to generate load against the target, the central server handles those incoming requests and sends responses back to the generator machines. After executing the command, the metrics appear in the generator consoles. This output is parsed and returned to PTR using an output processor. After the parsing is complete, PTR reports the metrics back to the end user.

Considering PTR's execution is intricate, we wanted to test the flow to ensure it was functional in the original state that we found it. Fastly was able to provide us with Google Cloud virtual machines so we could test that PTR was functional with the tools it originally supported. In the following sections, we explain what generators are, in terms of PTR, and the role that configuration files play.

4.2.1: Structure of generators in PTR

To provide Fastly with flexible load-testing scenarios and while also addressing K6's limitations, we built generators inside of PTR. The role of a generator is to implement the LoadGenerator interface, which "wraps" a load testing tool, indicating how the command is formatted and what to do after execution. The LoadGenerator interface structure and its three main functions are shown below in Figure 7.

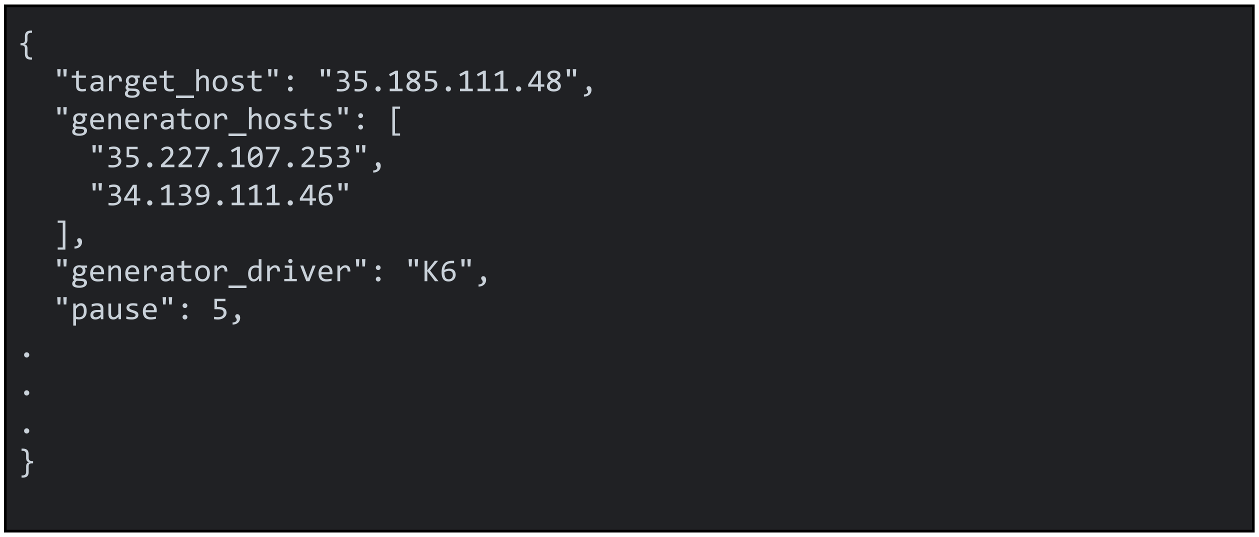


```
type LoadGenerator interface {
    Cmd(string) string
    OutputProcessor(io.Reader, string)
    PrintStats()
}
```

Figure 7: The LoadGenerator interface definition

The Cmd function builds the appropriate command string to run the tool based on options specified by the user. The OutputProcessor function is responsible for processing the resulting metrics from the tool's execution and PrintStats specifies how to print it to the user.

A key component for running these custom generators is the configuration file. Configuration files take the form of JSON objects containing parameters specific to the load-testing program, and are passed to the generator in order to construct the command string. Since PTR supports different types of load-testing generators, there are slight variations between the config files of different generator types; however, there are also fields that they all share in common. Figure 8 below shows an example of the parameters used across all config file types.



```
{
  "target_host": "35.185.111.48",
  "generator_hosts": [
    "35.227.107.253",
    "34.139.111.46"
  ],
  "generator_driver": "K6",
  "pause": 5,
.
.
.
}
```

Figure 8: Example of PTR config parameters shared across all generators

The “target_host” specifies the address of the machine that the generator hosts produce load against. This field is mandatory and only one target host can be specified. If no port is specified in the address, PTR uses the default port 22. The “generator_hosts” array is a list of virtual machines that generate load against the target machine. This field is mandatory and at least one generator host must be specified. The “generator_driver” is mandatory and specifies the type of generator (i.e. load-testing tool) that is used for that configuration. The possible options are “k6”, “wrk”, “idle-connections”, and “tls-perf” as these are the only generators currently supported. The “pause” field is optional and specifies a duration (in seconds) to wait after each scenario (in the scenario block) has finished running. In the following sections, we further expand on the configuration file options specific to each tool.

4.2.2: Integrating K6 into PTR

In order to integrate a K6 generator into PTR, it requires the ability to SCP files to multiple generators that are protected via a bastion host. As mentioned previously, each type of generator requires a slightly different configuration file. For each generator host to produce load using K6, the corresponding K6 JavaScript file must be available on the machine producing load. Using PTR’s K6 configuration file, a user can specify the K6 JavaScript files to share with each of the generator hosts. Figure 9 below is an example of how a user can structure their K6 configuration file to share files from PTR to each of the generator machines.

```
"files_to_generator_hosts": [
    {
      "local_path":
"/home/ubuntu/Desktop/ptr-mqp/internal/generators/K6/K6_js/test.js",
      "remote_addr": "/home/ubuntu/remote_jsfile.js",
      "file_permission": "0644"
    },
    {
      "local_path":
"/home/ubuntu/Desktop/ptr-mqp/internal/generators/K6/K6_js/test1.js",
      "remote_addr": "/home/ubuntu/remote_jsfile1.js",
      "file_permission": "0644"
    }
  ]
```

Figure 9: Structure of file_to_generator_hosts JSON object in K6 config file.

Within the "files_to_generator_hosts" JSON array, a user can specify objects that represent the K6 JS files. The "local_path" represents the local path of the K6 JS file on the machine executing PTR. The "remote_addr" represents the remote path of the K6 JS file once it has been copied from the machine executing PTR to the generator host. Lastly, the "file_permission" represents the permissions set on the file that is copied to the generator hosts. In the figure below "0644" gives both read and write access for the file specified in the "remote_addr".

As for the actual implementation, passing the K6 JavaScript files from PTR to the generator machines requires the use of an external golang SCP library called go-scp. This library allows machines to securely copy and share files to other machines over a SSH connection. However, at the enterprise level it is not safe to allow devices from external networks to form SSH sessions with machines on the private network. To protect internal infrastructure, companies implement a bastion network to serve as an interface between machines on the private network and those that are external. Figure 10 below shows an overview of how internal infrastructure on a private network can be protected via a bastion. Authenticating via a bastion provides a centralized location that all external network traffic must enter before gaining access to private resources on the internal network. To securely and properly SCP files into machines on Fastly's private network, a user must create an SSH session with the bastion. Assuming that the authentication is successful, PTR establishes a new SSH session from bastion to the generator machine. If this is successful then PTR shares the K6 JS file with the machines specified on the internal network thereby allowing traffic to be generated.

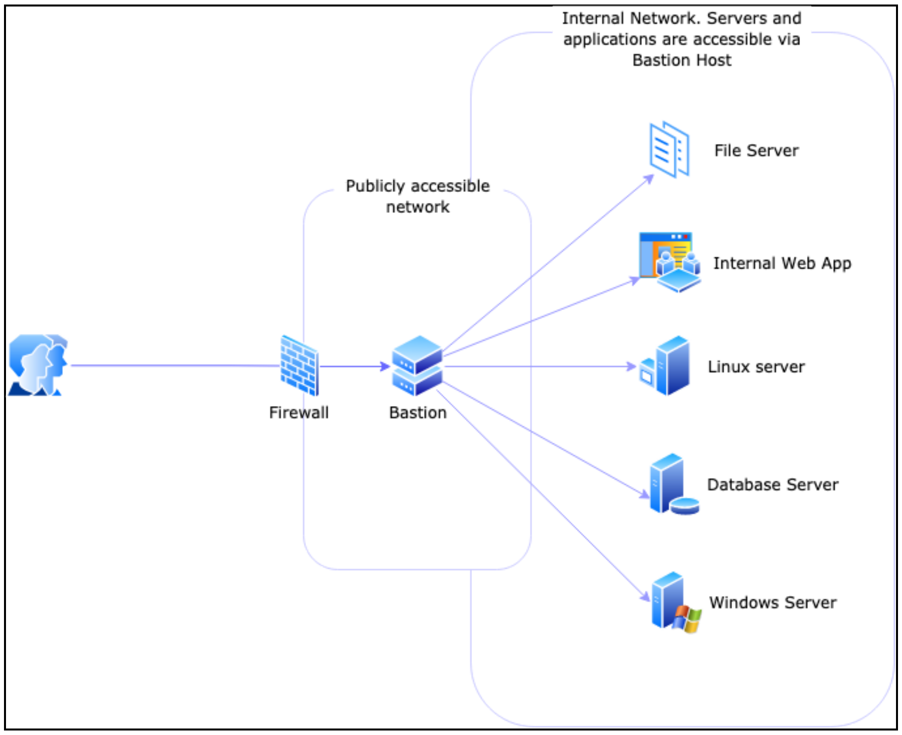
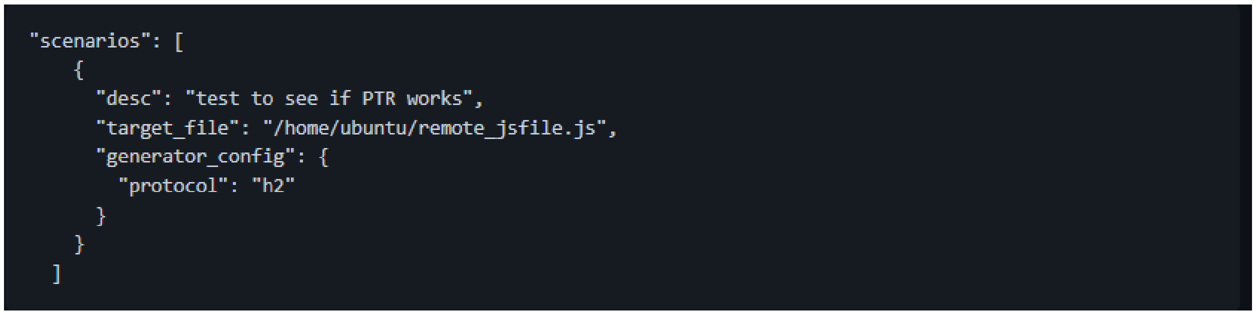

Internal Network. Servers and applications are accessible via Bastion Host
Publicly accessible network
File Server
Internal Web App
Firewall
Bastion
Linux server
Database Server
Windows Server

Figure 10: A diagram showing how a bastion host can be used to protect an internal network (*Remote Access Via Bastion Host*).

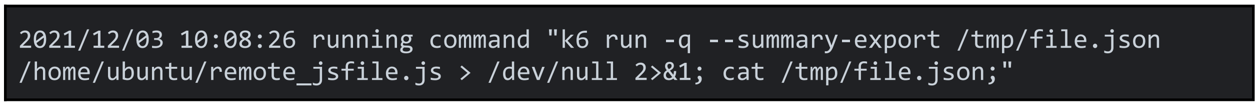
To tell PTR to run a specific K6 JS file on the generator host, the "scenarios" JSON array must be populated. Figure 11 below is an example of a single scenario within the "scenarios" JSON array. The "desc" represents a description of the test file the generator is set to run. The "target_file" represents the remote path of the K6 JS file that is run. Lastly, nested within the "generator_config" object is the "protocol" field. For the "protocol," a user can specify either "h1" or "h2," representing the HTTP protocol version in which the network traffic is sent over.



```
"scenarios": [
    {
      "desc": "test to see if PTR works",
      "target_file": "/home/ubuntu/remote_jsfile.js",
      "generator_config": {
        "protocol": "h2"
      }
    }
  ]
```

Figure 11: An example scenario from a PTR configuration file that runs K6

In order to execute a K6 load test for the scenario in Figure 11, Figure 12 shows the command that would be generated by PTR.

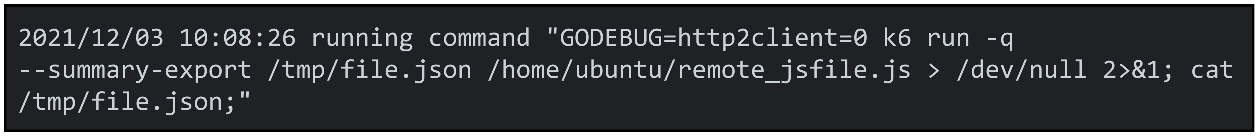


```
2021/12/03 10:08:26 running command "k6 run -q --summary-export /tmp/file.json
/home/ubuntu/remote_jsfile.js > /dev/null 2>&1; cat /tmp/file.json;"
```

Figure 12: PTR output showing the K6 command that runs on the generator hosts

The “--summary-export <file_path>” flag is responsible for outputting the metrics in a JSON object to the specified file path. In our case, we are placing the JSON file in the /tmp/ directory as it exists within all Linux machines. Following this, the command runs the “target_file” specified in the scenarios object as seen in Figure 11. In the event that HTTP requests fail, golang outputs the error messages into the console, making it difficult to parse the JSON object. Considering that failed requests already appear within the K6 metrics, we utilized the “/dev/null 2>&1” command to redirect all error messages to the /dev/null directory. Once K6 has completed its execution, the file.json that contains the metrics is outputted to the generator console. Finally, the console output is parsed and returned to PTR via the generator’s OutputProcessor.

In the event that the user specifies “h1” in the generator configuration, PTR generates a different command string. Figure 13 shows the new command executed by each generator host.



```
2021/12/03 10:08:26 running command "GODEBUG=http2client=0 k6 run -q
--summary-export /tmp/file.json /home/ubuntu/remote_jsfile.js > /dev/null 2>&1; cat
/tmp/file.json;"
```

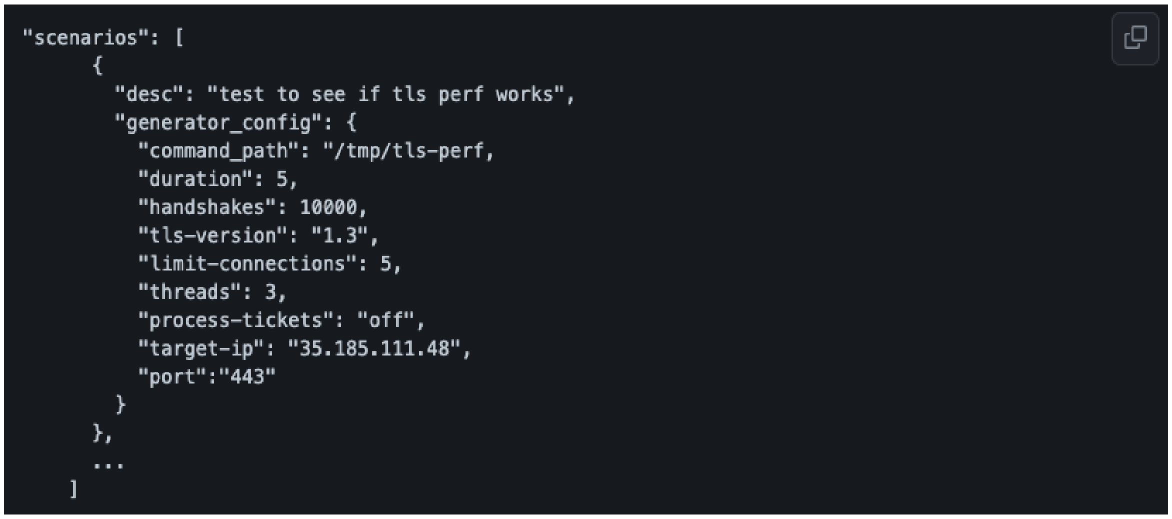
Figure 13: PTR output showing the K6 command that runs on the generator hosts when "h1" protocol is specified in the scenarios block.

As mentioned in section 4.1, in order to force HTTP/1.1 protocol, GODEBUG must set the http2client to 0. Giving the user this ability to specify a protocol helps address one of K6 limitations.

4.2.3: Integrating TLS-Perf into PTR

Considering K6 does not allow the user to control the TLS handshake rate, we implemented a generator for an open-source tool called TLS-Perf. By prompting "processor intensive cryptographic computations" on the server side, TLS-Perf works to stress-test the TLS handshake process. It does this by establishing a TLS handshake with an endpoint and then resetting the TCP connection without trying to transport data or perform a renegotiation (Krizhanovsky et al., 2020). With PTR, Fastly has the flexibility to use TLS-Perf in conjunction with K6 in order to test more advanced scenarios they were not previously able to run.

To run TLS-Perf using PTR, the user defines TLS-Perf specific configurations as a JSON object inside the scenarios array in a configuration file. For TLS-Perf, a majority of the options are set as a part of the "generator_config" that is nested in the JSON object. Figure 14 below specifies the possible fields that can be set in order to customize a TLS-Perf test scenario.

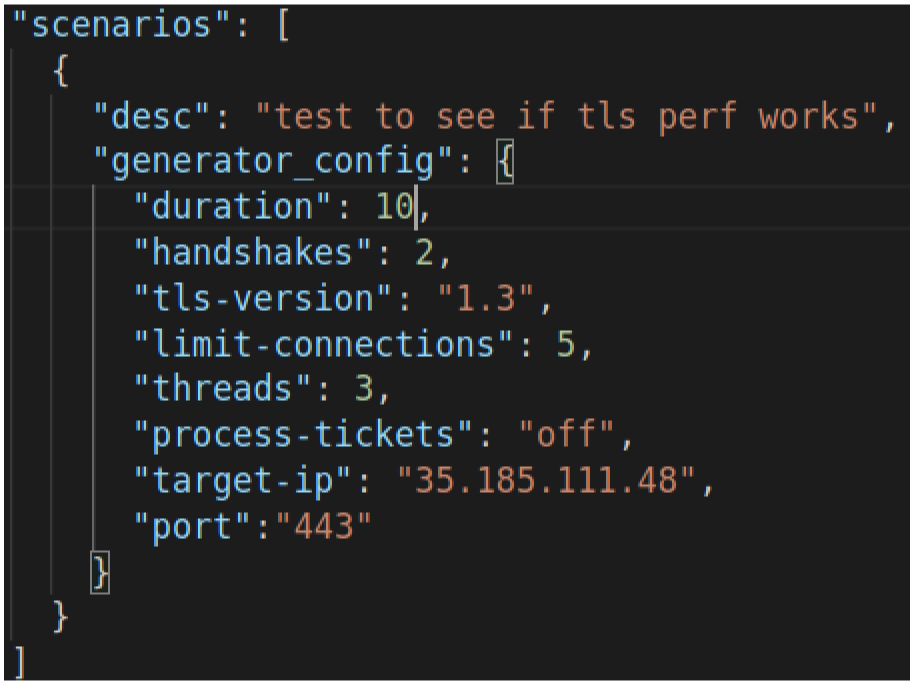


```
"scenarios": [
      {
        "desc": "test to see if tls perf works",
        "generator_config": {
          "command_path": "/tmp/tls-perf,
          "duration": 5,
          "handshakes": 10000,
          "tls-version": "1.3",
          "limit-connections": 5,
          "threads": 3,
          "process-tickets": "off",
          "target-ip": "35.185.111.48",
          "port":"443"
        }
      },
      ...
    ]
```

Figure 14: Screenshot of a sample scenario in a PTR configuration file that runs TLS-Perf

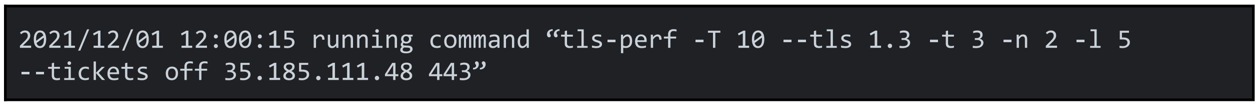
The “desc” field is optional and allows the user to enter a description of the scenario being run so that it can be printed in the console output when that scenario is being executed by PTR. The fields that can be specified inside of the “generator_config” object are command path, duration, handshakes, TLS version, limit connections, threads, process tickets, target ip, and port. The “command_path” field is optional and allows the user to specify the path to where the TLS-Perf binaries are located on the generator machines. By default, the TLS-Perf generator prefixes the command string with the name of the binary, assuming that the file path of where the binary is located is a part of the $PATH environment variable. In order to successfully run TLS-Perf using PTR, the user needs to specify at least the “duration” or “handshakes” field. The “duration” field specifies how long to generate TLS handshakes for and the “handshakes” field specifies how many handshakes to generate against the target. If neither is specified then PTR reports an error, and if both are specified then TLS-Perf gives handshakes a higher priority. The “tls-version” field is optional and allows the user to specify which TLS version to use when generating TLS handshakes. The possible options for this field are “1.2”, “1.3”, and “any”. If nothing is specified, TLS-Perf uses version 1.2 and if “any” is specified then TLS-Perf uses both versions to create TLS handshakes. The “limit-connections”, “threads”, and “process-tickets” fields are also optional. The “limit-connections” field specifies the limit of parallel connections to use for each thread, the “threads” field specifies the number of threads to create in order to generate TLS handshakes, and the “process-tickets” field allows the user to specify whether session resumption is enabled or not. The possible options for the “process-tickets” field are “on”, “off”, and “advertise”. Finally, the “target-ip” and “port” fields are both mandatory. These specify the target IP address and the port of the server you would like to target, respectively. It is important to note that the port is a separate field and should not be specified as part of the target IP.

The options specified in the configuration file allows the TLS-Perf generator to create the command to be run on the generator virtual machines. Figure 15 below shows another sample scenario object and Figure 16 shows the command string that is printed out to the console as it is being executed on the generator machines.



```
"scenarios": [
  {
    "desc": "test to see if tls perf works",
    "generator_config": {
      "duration": 10,
      "handshakes": 2,
      "tls-version": "1.3",
      "limit-connections": 5,
      "threads": 3,
      "process-tickets": "off",
      "target-ip": "35.185.111.48",
      "port":"443"
    }
  }
]
```

Figure 15: Screenshot of another sample TLS-Perf scenario in a PTR configuration file



```
2021/12/01 12:00:15 running command “tls-perf -T 10 --tls 1.3 -t 3 -n 2 -l 5
--tickets off 35.185.111.48 443”
```

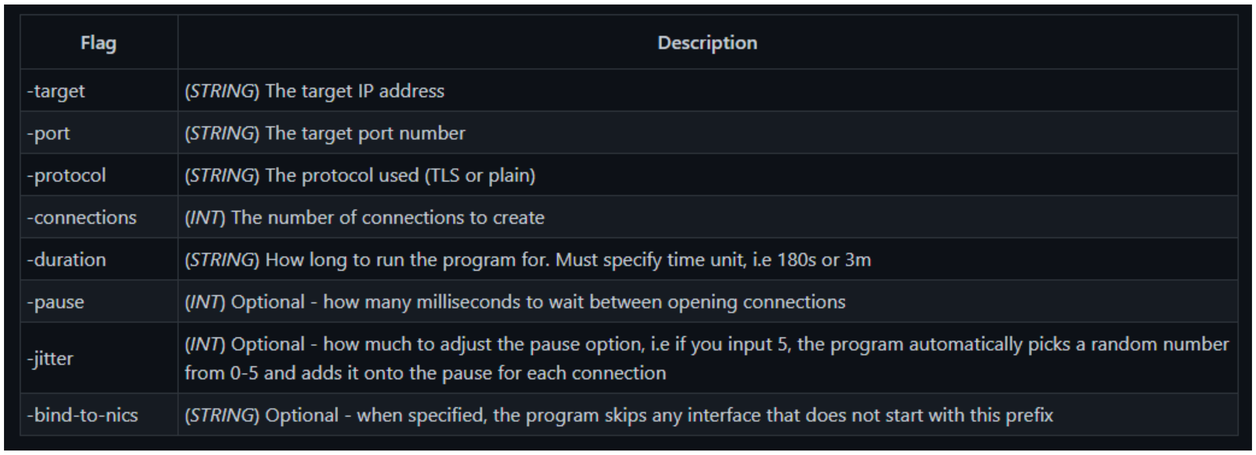
Figure 16: Screenshot of the command string that is generated from running PTR with the scenario object seen in Figure 15

4.2.4: Integrating the Idle-Connections program into PTR

Another useful feature we integrated into PTR was a program to generate idle connections against a target host. An idle connection is a connection that is opened on the server without issuing any requests and remains alive until the server closes it. As an example, this would be the equivalent of a user opening up a webpage but not interacting with it at all, just keeping the page running. The ability to generate idle connections like this would be beneficial to Fastly's load testing needs in particular because of a recent memory consumption issue they were experiencing with h2o, their HTTP server. Specifically, when they modified h2o to increase the default response buffer size from 4kb to 1mb, these 1mb buffers were allocated even for idle connections. This is an issue because idle connections by nature do not issue any HTTP requests, and so millions of clients connecting to a single server instance consumes much more memory than before. Previously, there was no way of replicating the scenario and testing it. The idle connections program helps solve this problem since it is able to generate millions of TLS connections to the TLS terminator, so we are able to see exactly when and where the memory leakage occurs.

To implement the idle connections feature within PTR, we first had to build a standalone program in Golang that could be configured to run with different parameters on the command line. At a high level, the program iterates through each available network interface controller (NIC) on the generator host and creates a specified number of connections to a server and port (via TLS or plaintext) for a set duration per NIC. Each connection is created using its own separate goroutine (i.e, Golang's implementation of a process thread) so that the program may run concurrently; however, because systems usually have a default limit of about 6000 open file connections that can run at a time, the program also checks for sufficient file descriptors prior to execution. This prevents a panic error and allows the user to increase file descriptors if needed. Each time the program detects that the server has terminated connections, it spawns new ones. If an individual connection's deadline expires (i.e when the connection times out), it extends the deadline. This continues until the duration has finished, at which point the program fully exits, outputting a JSON in the console with a few basic stats: Total Connections Opened, Total Timeouts, and Average Time to Connect to Server.

The complete set of flags that the program takes in along with their descriptions can be seen below in Table 5.



Flag	Description
-target	(*STRING*) The target IP address
-port	(*STRING*) The target port number
-protocol	(*STRING*) The protocol used (TLS or plain)
-connections	(*INT*) The number of connections to create
-duration	(*STRING*) How long to run the program for. Must specify time unit, i.e 180s or 3m
-pause	(*INT*) Optional - how many milliseconds to wait between opening connections
-jitter	(*INT*) Optional - how much to adjust the pause option, i.e if you input 5, the program automatically picks a random number from 0-5 and adds it onto the pause for each connection
-bind-to-nics	(*STRING*) Optional - when specified, the program skips any interface that does not start with this prefix

Table 5: Table describing the flags that can be passed into idle_connections

As shown, there are three optional flags: pause, jitter, and bind-to-nics. The pause and jitter options allow the user to better see the effect of opening each connection because there is a more visible time gap between them. The bind-to-nics flag is an option specific to Fastly; when testing from an internal Fastly address, only certain NICs should be accessed because some are restricted management IP addresses. When the user specifies a prefix for the bind-to-nics flag, it only accesses interfaces starting with this string, effectively avoiding any management networks.

To integrate this program into PTR, we cloned the idle connections program onto each of the host machines so that they could spawn connections at the same time. As with K6 and TLS-Perf, a config file specific to idle connections is necessary to run it properly via PTR. Figure 17 below shows the scenarios array with the following parameters.

```
"scenarios": [
       {
           "desc": "test to see if idleconns works",
           "generator_config": {
               "target": "35.185.111.48",
               "port": "443",
               "protocol": "TLS",
               "connections": 100,
               "duration": "30s"
               "pause": 5,
               "jitter": 10,
               "bind-to-nics": "vlan"
           }
       }
   ]
```

Figure 17: The structure and parameters used in the TLS-perf config file.

As shown, the array uses all of the parameters the idle connections program can take in and builds a string that runs on the command line. With the above example config file, the generated command would look like the one shown in Figure 18 below.

```
idle_connections -target 35.185.111.48 -port 443 -protocol TLS -connections 100
-duration 30s -pause 5 -jitter 10 -bind-to-nics vlan
```

Figure 18: Command string generated from idle-connections generator config file

This command runs the program, opening 100 connections against the target IP 35.185.111.48 on port 443 and it is using the TLS protocol to initiate a handshake per connection. Because we specified 5 for pause and 10 for jitter, the program waits at least 5 milliseconds between opening each connection and adds a random number of milliseconds in the range 0-10 each time. Finally, because we specified vlan as the bind-to-nics prefix, the program only loops through interfaces starting with vlan (i.e. vlan100, vlan200, etc.).

4.3: Running multiple generators concurrently

Given that PTR's key feature is wrapping multiple types of load-testing tool, our next logical step was to extend support for running these generators simultaneously. To implement this change, we altered the PTR command string to allow the user to input multiple comma-separated config files via the --test-config flag. Then, we modified the code in main.go to iterate through each of these user-specified config files (rather than just running a singular config file by default) and create a separate goroutine process for each one so the program runs concurrently. This way, we are able to widen the scope of PTR by running multiple load-testing tools that target different areas of Fastly's system at once. The next chapter further expands on how we used these extended features and the discoveries made by running PTR against internal Fastly infrastructure.

CHAPTER 5: Evaluation

This chapter evaluates PTR as a load-testing multi-tool and describes our key findings from testing Performance Test Runner (PTR). Section 5.1 discusses what PTR offers Fastly and the improvement over their previous load-testing technology. Section 5.2 dives into tests and discoveries made while using PTR against Fastly production machines.

5.1: Leveraging PTR capabilities

With PTR, Fastly's load testing capabilities have significantly improved in comparison to the range of tests they were able to run with H2Load. As mentioned previously, PTR's primary advantage is its ability to wrap various tools. This in itself was a substantial upgrade from Varnish Load Tester (VLT), because we were able to implement specialized programs like K6, TLS-perf, and Idle Connections within PTR that fulfilled Fastly's requirements. In the following sections, we further detail how we can leverage these features of PTR to Fastly's advantage and the insights they can provide.

In order to run PTR, the user must specify a command string along with a set of arguments. An example command string is shown below in Figure 19.

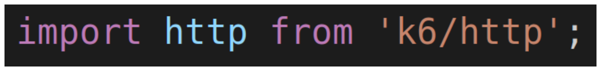
```
ptr --test-config test-k6.json --user ubuntu --ssh-keyfile
/home/ubuntu/.ssh/id_rsa --bastion-host 34.138.75.34
```

Figure 19: Example command string used to run PTR

The "--test-config" is a mandatory flag that allows the user to specify the path to the configuration file they would like to load into PTR. Here, we pass in the file "test-k6.json" which specifies to run the K6 generator. The "user" flag specifies the username of the virtual machines that PTR needs to SSH into. "--ssh-keyfile" is where the user must pass in the path to their SSH private key for authentication. Finally, the "--bastion-host" flag, as mentioned in Section 4.2.2, allows the user to specify the address of the bastion that they use to establish a secure connection to the generator machines. In this case, PTR uses the bastion host 34.138.75.35 and creates an SSH connection to whichever generator host VMs are specified in test-k6.json.

5.1.1: Customization offered by K6 scripts

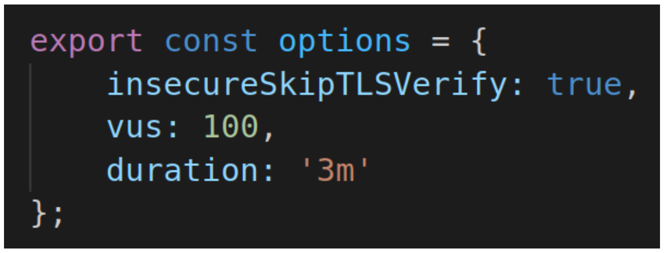
As mentioned in previous chapters, K6 allows the user to specify a test scenario in a JavaScript configuration file. A basic K6 JavaScript configuration file is composed of three different parts: the import statement, the options object, and the default function. Every file must have the correct K6 import at the top of the file as seen in Figure 20 below.



```
import http from 'k6/http';
```

Figure 20: Import statement that must be included at the top of a K6 JavaScript configuration file

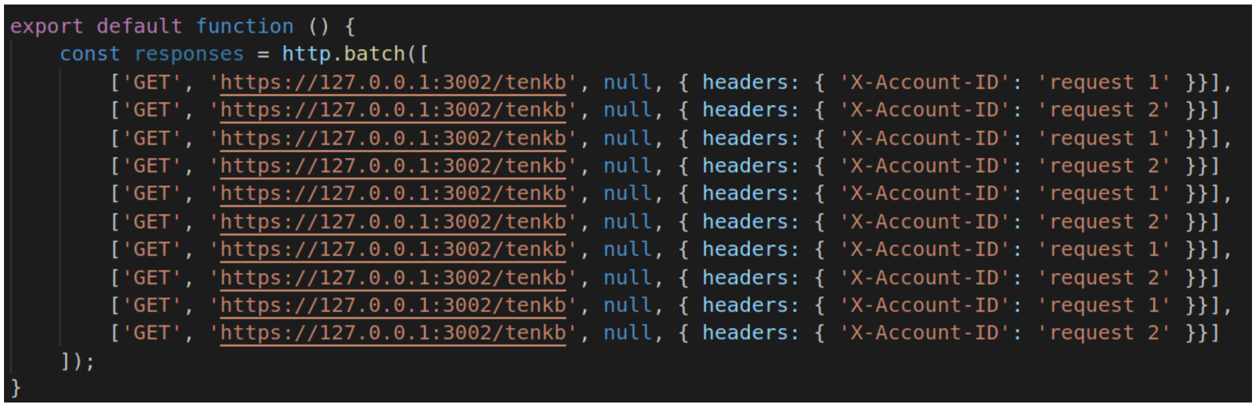
By importing the k6/http module, the user has access to the HTTP API that contains functionality for performing HTTP transactions. The second component of a K6 configuration file is the options object seen below in Figure 21.



```
export const options = {
    insecureSkipTLSVerify: true,
    vus: 100,
    duration: '3m'
};
```

Figure 21: Options object that is used to describe the test scenario that is run

The options object in a configuration file describes the different settings to use when running a specific test scenario. The options seen in Figure 21 above specify to run a test for a duration of three minutes, with one hundred virtual users, while ignoring the verification of TLS certificates. The “insecureSkipTLSVerify” option was set to true for all of the tests we ran considering the central server that we set up had self-signed TLS certificates. If the option was removed, we would get errors when issuing requests over HTTPS because the certificates cannot be verified by an official certificate authority. Finally, the last component that a K6 JavaScript configuration file must have is a default function. The default function is the entry point for each virtual user that is created for a test scenario, and works similar to a main function in a program (*Test life cycle, n.d.*). Each virtual user is essentially a thread that runs the code inside of the default function and this is used to simulate real users in a production setting. Figure 22 below shows an example default function that specifies ten HTTP GET requests to execute.

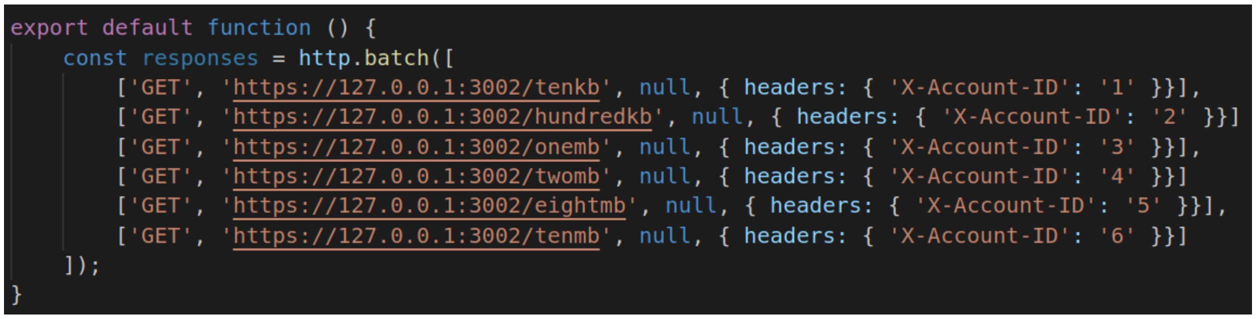


```
export default function () {
    const responses = http.batch([
        ['GET', 'https://127.0.0.1:3002/tenkb', null, { headers: { 'X-Account-ID': 'request 1' }}],
        ['GET', 'https://127.0.0.1:3002/tenkb', null, { headers: { 'X-Account-ID': 'request 2' }}]
        ['GET', 'https://127.0.0.1:3002/tenkb', null, { headers: { 'X-Account-ID': 'request 1' }}],
        ['GET', 'https://127.0.0.1:3002/tenkb', null, { headers: { 'X-Account-ID': 'request 2' }}]
        ['GET', 'https://127.0.0.1:3002/tenkb', null, { headers: { 'X-Account-ID': 'request 1' }}],
        ['GET', 'https://127.0.0.1:3002/tenkb', null, { headers: { 'X-Account-ID': 'request 2' }}]
        ['GET', 'https://127.0.0.1:3002/tenkb', null, { headers: { 'X-Account-ID': 'request 1' }}],
        ['GET', 'https://127.0.0.1:3002/tenkb', null, { headers: { 'X-Account-ID': 'request 2' }}]
        ['GET', 'https://127.0.0.1:3002/tenkb', null, { headers: { 'X-Account-ID': 'request 1' }}],
        ['GET', 'https://127.0.0.1:3002/tenkb', null, { headers: { 'X-Account-ID': 'request 2' }}]
    ]);
}
```

Figure 22: Example default function that specifies ten GET requests to be executed

By calling http.Batch, the user is able to specify an array or "batch" of requests to be sent out concurrently. Each request in the batch array accepts four arguments: HTTP method, URL, body, and "params". The HTTP method and URL are both required fields whereas body and "params" are optional. The HTTP method refers to the HTTP method of the request to be sent. The URL argument refers to the URL that the request is issued to. The body represents the body to be sent along with the request. The "params" field gives the user the ability to specify details like headers and tags. For example, in the case that the user would like to specify headers in the params field but nothing in the body, they would set the body parameter to null as seen in Figure 22 above.

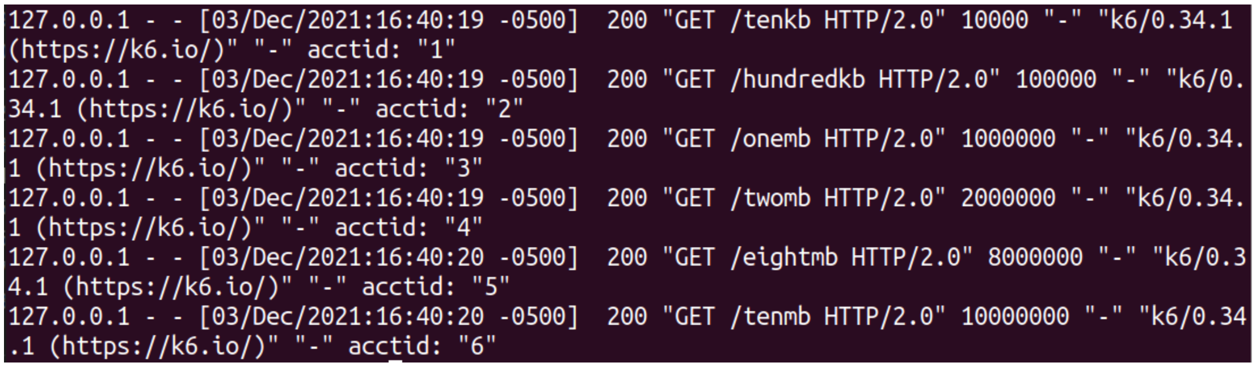
With K6 now integrated into PTR, Fastly can utilize K6's ability to customize requests that are sent out in a single test run. Two of the optional requirements that Fastly found beneficial to have in a load-testing tool was the ability to send requests targeting different URLs and issuing requests with different headers in a single test run. Figure 23 below shows a K6 batch of requests targeting different files from a local IP address with unique headers.



```
export default function () {
    const responses = http.batch([
        ['GET', 'https://127.0.0.1:3002/tenkb', null, { headers: { 'X-Account-ID': '1' }}],
        ['GET', 'https://127.0.0.1:3002/hundredkb', null, { headers: { 'X-Account-ID': '2' }}]
        ['GET', 'https://127.0.0.1:3002/onemb', null, { headers: { 'X-Account-ID': '3' }}],
        ['GET', 'https://127.0.0.1:3002/twomb', null, { headers: { 'X-Account-ID': '4' }}]
        ['GET', 'https://127.0.0.1:3002/eightmb', null, { headers: { 'X-Account-ID': '5' }}],
        ['GET', 'https://127.0.0.1:3002/tenmb', null, { headers: { 'X-Account-ID': '6' }}]
    ]);
}
```

Figure 23: Default function that specifies six GET requests to different URLs

By modifying the URL parameter in each request, we are able to instruct K6 to have the virtual users send each of the requests specified. The user is also able to modify the headers set in the params field of a request to customize the headers that are sent with each request. When running K6 with a JavaScript file that includes the default function seen in Figure 23, each of the requests successfully reaches the NGINX server we set up for testing, as seen in Figure 24 below.

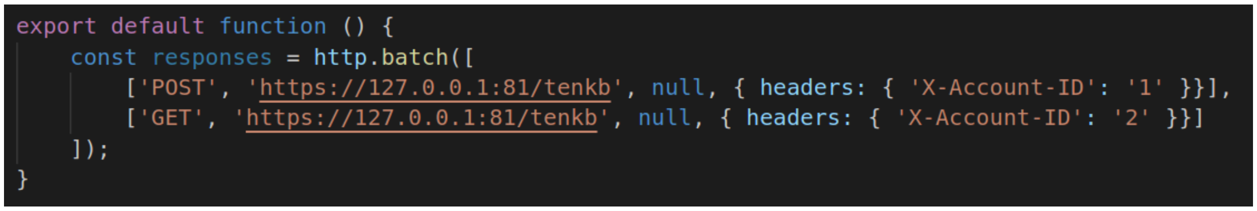


```
127.0.0.1 - - [03/Dec/2021:16:40:19 -0500]  200 "GET /tenkb HTTP/2.0" 10000 "-" "k6/0.34.1
(https://k6.io/)" "-" acctid: "1"
127.0.0.1 - - [03/Dec/2021:16:40:19 -0500]  200 "GET /hundredkb HTTP/2.0" 100000 "-" "k6/0.
34.1 (https://k6.io/)" "-" acctid: "2"
127.0.0.1 - - [03/Dec/2021:16:40:19 -0500]  200 "GET /onemb HTTP/2.0" 1000000 "-" "k6/0.34.
1 (https://k6.io/)" "-" acctid: "3"
127.0.0.1 - - [03/Dec/2021:16:40:19 -0500]  200 "GET /twomb HTTP/2.0" 2000000 "-" "k6/0.34.
1 (https://k6.io/)" "-" acctid: "4"
127.0.0.1 - - [03/Dec/2021:16:40:20 -0500]  200 "GET /eightmb HTTP/2.0" 8000000 "-" "k6/0.3
4.1 (https://k6.io/)" "-" acctid: "5"
127.0.0.1 - - [03/Dec/2021:16:40:20 -0500]  200 "GET /tenmb HTTP/2.0" 10000000 "-" "k6/0.34
.1 (https://k6.io/)" "-" acctid: "6"
```

Figure 24: Access log for NGINX server showing the incoming requests targeting different files

The access log for an NGINX server reports incoming requests and other information about the requests as seen above. In Figure 24, we can see that there are six requests that were handled by the server and it reports that there was a single request for tenkb, hundredkb, onemb, twomb, eightmb, and tenmb. When observing the timestamps of the access log, we can see that all of the requests were issued and handled at the exact same moment with there being only a one millisecond difference between the first four requests and the last two. This demonstrates K6 being able to accurately model real-world traffic on a server where requests are coming in at the same time rather than one by one. These different file names represent zero-filled objects of different sizes that we used for testing. We can see that the status for each of these requests is 200, which means they were all successfully handled by the server. We can also see that the headers that were found in the incoming requests match the headers that were specified in the K6 JavaScript configuration file.

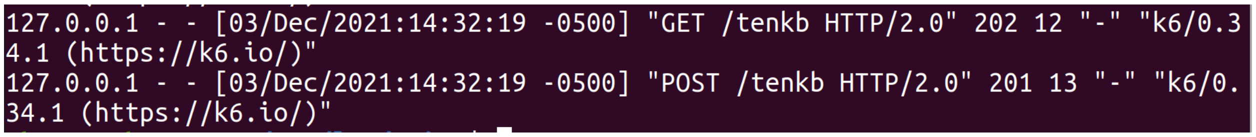
K6 also gives Fastly the flexibility to issue requests with different HTTP methods. By being able to run a test scenario with a variety of HTTP methods, Fastly can more accurately simulate production scenarios. Figure 25 below shows two requests in a K6 batch with different HTTP methods.



```
export default function () {
    const responses = http.batch([
        ['POST', 'https://127.0.0.1:81/tenkb', null, { headers: { 'X-Account-ID': '1' }}],
        ['GET', 'https://127.0.0.1:81/tenkb', null, { headers: { 'X-Account-ID': '2' }}]
    ]);
}
```

Figure 25: Default function that specifies a POST request and GET request in the same run

When running K6 with a JavaScript file that includes the default function seen in Figure 25 above, the access log in Figure 26 reports the HTTP method of the incoming requests.



```
127.0.0.1 - - [03/Dec/2021:14:32:19 -0500] "GET /tenkb HTTP/2.0" 202 12 "-" "k6/0.3
4.1 (https://k6.io/)"
127.0.0.1 - - [03/Dec/2021:14:32:19 -0500] "POST /tenkb HTTP/2.0" 201 13 "-" "k6/0.
34.1 (https://k6.io/)"
```

Figure 26: Access log for NGINX server showing the incoming requests with different HTTP methods

By looking at the access log we can verify that K6 allows the user to customize the HTTP method for each request in a batch.

5.1.2: Handling K6 generator output

A helpful feature we extended in PTR was support for parsing K6 generator output. Compared to other load-testing tools we researched, K6 provides the user with verbose output covering a wide range of network performance metrics. K6 supports four types of metrics: counter, gauge, rate, and trend. Below are descriptions of each category of metrics:

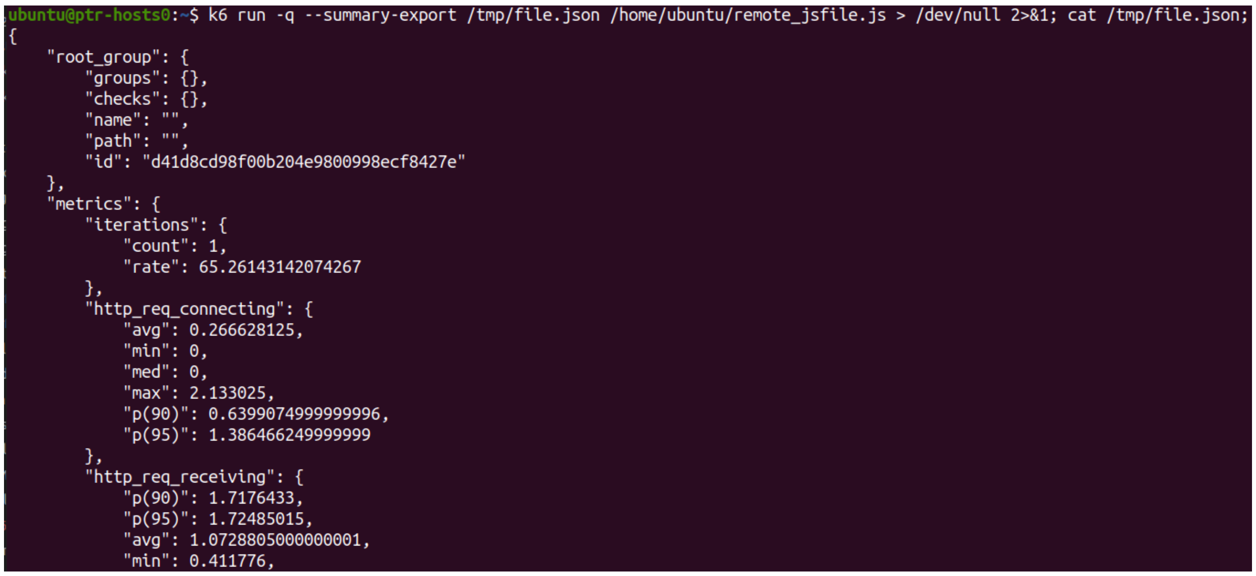
- Counter metrics cumulatively sum up values
- Gauge metrics store the minimum, maximum, and most recent values added
- Rate metrics track the percentage of non-zero values added
- Trend metrics provide extensive statistics such as minimum, maximum, average, and percentiles for the added values.

Table 6 below displays the metrics that K6 collects along with the category of each metric.

Metric Name	Type	Description
“http_reqs”	Counter	HTTP requests generated, in total
“data_received”	Counter	The amount of data received
“data_sent”	Counter	The amount of data sent
“http_req_blocked”	Trend	Time spent blocked before issuing the request (while waiting for a free TCP connection slot)
“http_req_connecting”	Trend	Time spent establishing connection to the host
“http_req_tls_handshaking”	Trend	Time spent TLS handshaking with the host
“http_req_sending”	Trend	Time spent sending data to the host
“http_req_waiting”	Trend	Time spent waiting for response from the host, this is typically known as “time to first byte” or TTFB
“http_req_receiving”	Trend	Time spent receiving data from the host
“http_req_duration”	Trend	Total time for the request. The sum of sending time, waiting time, and receiving time
“iteration_duration”	Trend	The amount of time it took to execute one full iteration of the default function
“http_req_failed”	Rate	Rate of failed requests

Table 6: Built-in metrics provided by K6 (*Metrics*, n.d.). Due to the amount of metrics, certain metrics have been left out (see Appendix A for the full table)

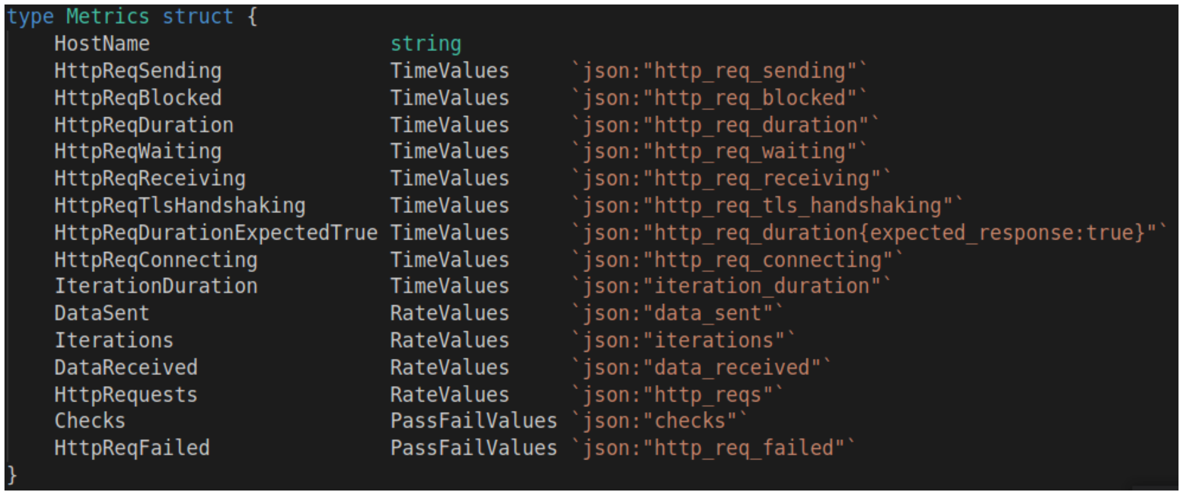
As mentioned in previous chapters, when PTR creates a thread to SSH into the generator machines and executes the generated command string, the remote machines print out a JSON object containing all of the metrics provided by K6. Figure 27 below shows what it would look like if a user ran the generated command string directly on the generator machines.



```
ubuntu@ptr-hosts0:~$ k6 run -q --summary-export /tmp/file.json /home/ubuntu/remote_jsfile.js > /dev/null 2>&1; cat /tmp/file.json;
{
    "root_group": {
        "groups": {},
        "checks": {},
        "name": "",
        "path": "",
        "id": "d41d8cd98f00b204e9800998ecf8427e"
    },
    "metrics": {
        "iterations": {
            "count": 1,
            "rate": 65.26143142074267
        },
        "http_req_connecting": {
            "avg": 0.266628125,
            "min": 0,
            "med": 0,
            "max": 2.133025,
            "p(90)": 0.6399074999999996,
            "p(95)": 1.386466249999999
        },
        "http_req_receiving": {
            "p(90)": 1.7176433,
            "p(95)": 1.72485015,
            "avg": 1.0728805000000001,
            "min": 0.411776,
```

Figure 27: Screenshot showing the JSON object containing K6 metrics being printed out in a generator machine

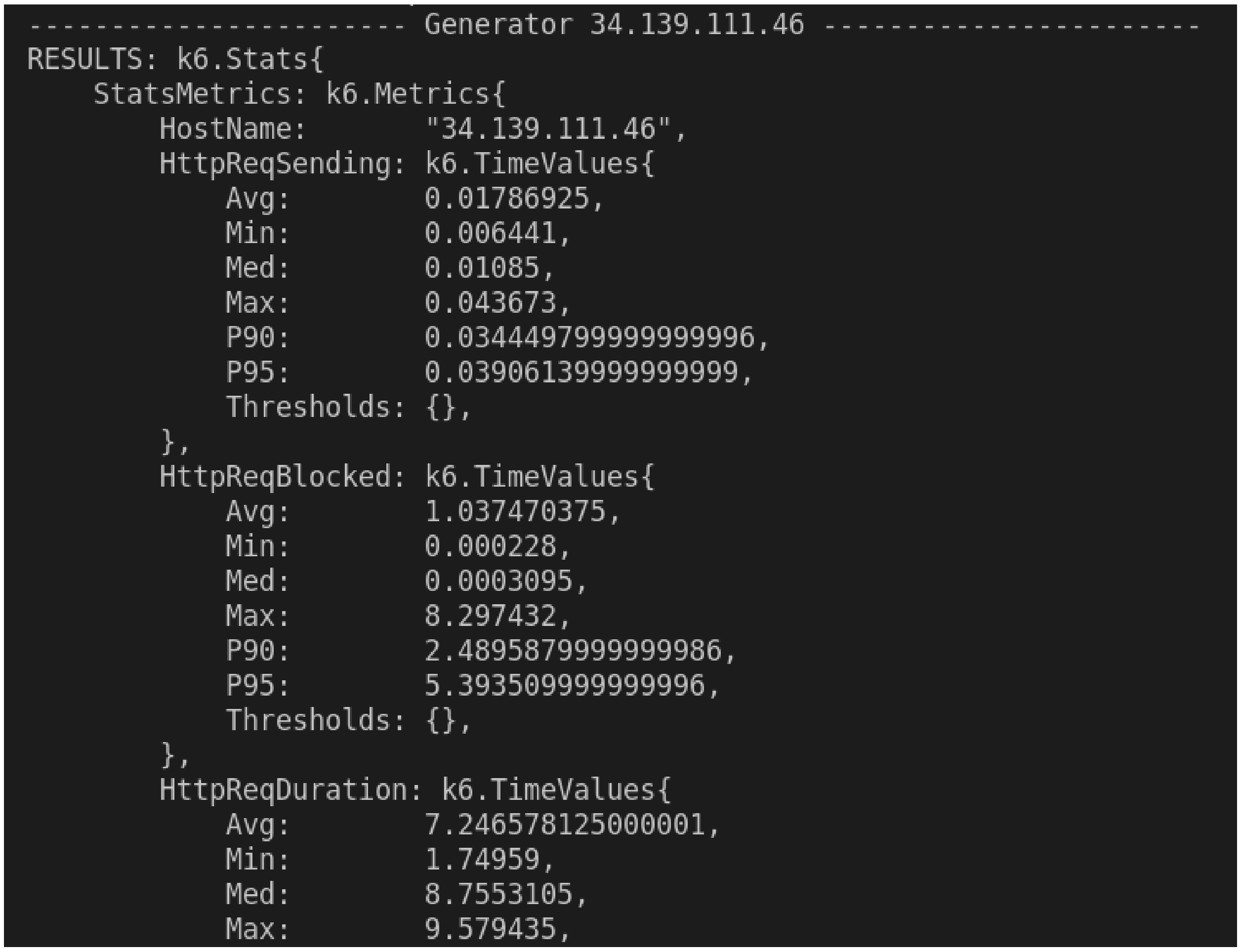
Although this is what it looks like when the PTR generated command string runs directly on a generator host, the user would never see this because it is handled in the background. Once the OutputProcessor has read in a JSON object, it then unmarshalls it into a struct that we created inside of the K6 PTR generator. Figure 28 below shows the structure of the struct that hold the metrics.



```
type Metrics struct {
    HostName                    string
    HttpReqSending              TimeValues     `json:"http_req_sending"`
    HttpReqBlocked              TimeValues     `json:"http_req_blocked"`
    HttpReqDuration             TimeValues     `json:"http_req_duration"`
    HttpReqWaiting              TimeValues     `json:"http_req_waiting"`
    HttpReqReceiving            TimeValues     `json:"http_req_receiving"`
    HttpReqTlsHandshaking       TimeValues     `json:"http_req_tls_handshaking"`
    HttpReqDurationExpectedTrue TimeValues     `json:"http_req_duration{expected_response:true}"`
    HttpReqConnecting           TimeValues     `json:"http_req_connecting"`
    IterationDuration           TimeValues     `json:"iteration_duration"`
    DataSent                    RateValues     `json:"data_sent"`
    Iterations                  RateValues     `json:"iterations"`
    DataReceived                RateValues     `json:"data_received"`
    HttpRequests                RateValues     `json:"http_reqs"`
    Checks                      PassFailValues `json:"checks"`
    HttpReqFailed               PassFailValues `json:"http_req_failed"`
}
```

Figure 28: Structure of the struct that holds the metrics in the JSON object printed at the end of K6's execution

The metrics that are stored in the "Metrics" struct contain other nested structs that hold the specific statistics related to each metric such as average, minimum, and count. Although not all of the metrics provided by K6 are included in the struct, modifying it to include other metrics is straightforward. Our team chose to display the metrics that would be most beneficial to a Fastly user running load-tests. When a K6 scenario has finished running, the end-user sees all of the metrics printed to the console with the generator printed at the top of each generator's output, as seen in Figure 29.



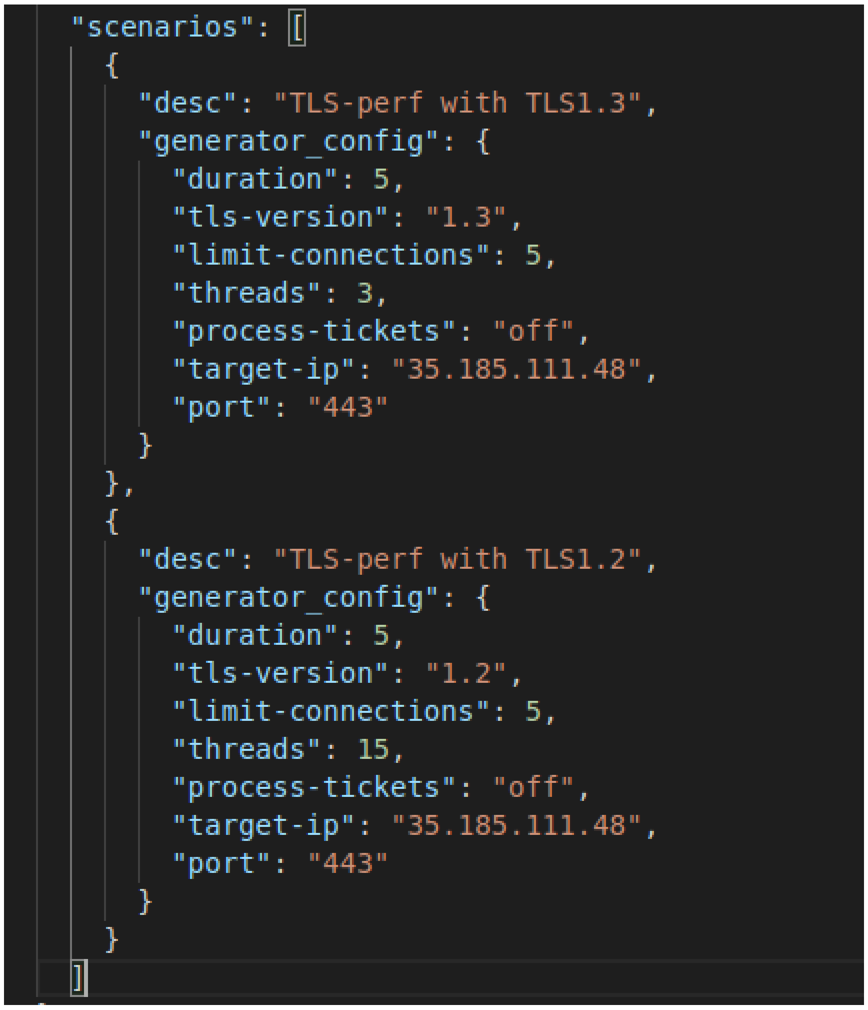
```
---------------------- Generator 34.139.111.46 ----------------------
RESULTS: k6.Stats{
    StatsMetrics: k6.Metrics{
        HostName:       "34.139.111.46",
        HttpReqSending: k6.TimeValues{
            Avg:        0.01786925,
            Min:        0.006441,
            Med:        0.01085,
            Max:        0.043673,
            P90:        0.034449799999999996,
            P95:        0.03906139999999999,
            Thresholds: {},
        },
        HttpReqBlocked: k6.TimeValues{
            Avg:        1.037470375,
            Min:        0.000228,
            Med:        0.0003095,
            Max:        8.297432,
            P90:        2.4895879999999986,
            P95:        5.393509999999996,
            Thresholds: {},
        },
        HttpReqDuration: k6.TimeValues{
            Avg:        7.246578125000001,
            Min:        1.74959,
            Med:        8.7553105,
            Max:        9.579435,
```

Figure 29: Screenshot of PTR output when a K6 scenario has finished running on a generator host (see Appendix B for the full output)

Considering a user may specify multiple generator hosts, it was crucial that we printed the name/IP address of the generator host so that the user is able to navigate the output with ease.

5.1.3: Running multiple scenarios from single configuration file

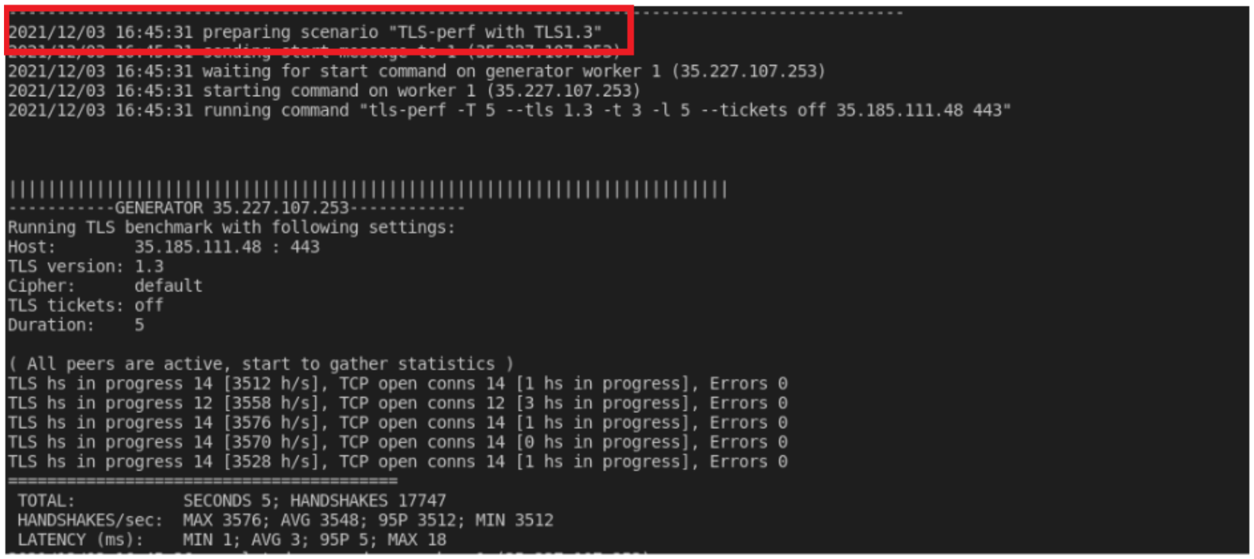
The ability to run multiple scenarios per PTR execution is another essential part of the tool's custom load-testing. By not having to re-run PTR, the user is able to run more tests in a shorter amount of time and better track the server side performance over a timeframe. Figure 30 shows how the user would specify multiple scenarios via the config file.



```
"scenarios": [
  {
    "desc": "TLS-perf with TLS1.3",
    "generator_config": {
      "duration": 5,
      "tls-version": "1.3",
      "limit-connections": 5,
      "threads": 3,
      "process-tickets": "off",
      "target-ip": "35.185.111.48",
      "port": "443"
    }
  },
  {
    "desc": "TLS-perf with TLS1.2",
    "generator_config": {
      "duration": 5,
      "tls-version": "1.2",
      "limit-connections": 5,
      "threads": 15,
      "process-tickets": "off",
      "target-ip": "35.185.111.48",
      "port": "443"
    }
  }
]
```

Figure 30: Image of config file that specifies multiple scenarios

As seen in Figure 30, by listing multiple scenarios it is more efficient for the users to run tests one after another rather than running an instance of PTR per scenario. The TLS-Perf scenario with 3 threads runs first for a duration of 5 seconds and then the next scenario with 15 threads runs for a duration of 5 seconds. Figures 31 and 32 show the output in PTR after running multiple scenarios.



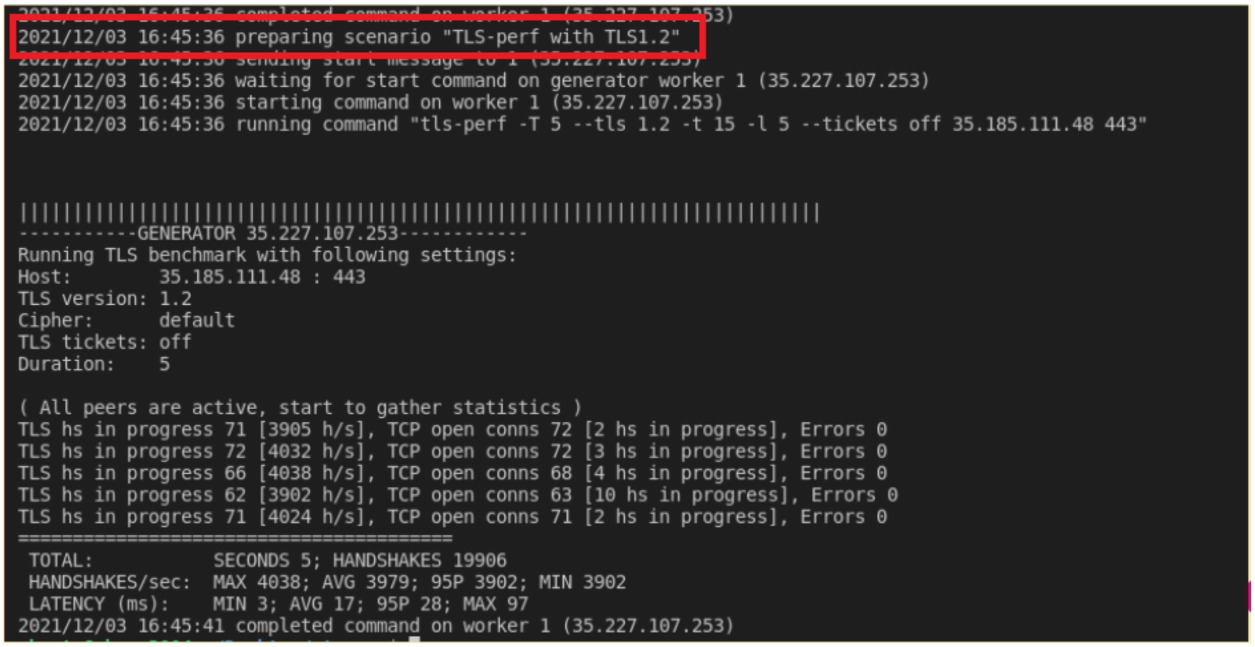
```
---------------------------------------------------------------------------------------------
2021/12/03 16:45:31 preparing scenario "TLS-perf with TLS1.3"
2021/12/03 16:45:31 sending start message to 1 (35.227.107.253)
2021/12/03 16:45:31 waiting for start command on generator worker 1 (35.227.107.253)
2021/12/03 16:45:31 starting command on worker 1 (35.227.107.253)
2021/12/03 16:45:31 running command "tls-perf -T 5 --tls 1.3 -t 3 -l 5 --tickets off 35.185.111.48 443"



||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
-----------GENERATOR 35.227.107.253-----------
Running TLS benchmark with following settings:
Host:        35.185.111.48 : 443
TLS version: 1.3
Cipher:      default
TLS tickets: off
Duration:    5

( All peers are active, start to gather statistics )
TLS hs in progress 14 [3512 h/s], TCP open conns 14 [1 hs in progress], Errors 0
TLS hs in progress 12 [3558 h/s], TCP open conns 12 [3 hs in progress], Errors 0
TLS hs in progress 14 [3576 h/s], TCP open conns 14 [1 hs in progress], Errors 0
TLS hs in progress 14 [3570 h/s], TCP open conns 14 [0 hs in progress], Errors 0
TLS hs in progress 14 [3528 h/s], TCP open conns 14 [1 hs in progress], Errors 0
========================================
 TOTAL:           SECONDS 5; HANDSHAKES 17747
 HANDSHAKES/sec:  MAX 3576; AVG 3548; 95P 3512; MIN 3512
 LATENCY (ms):    MIN 1; AVG 3; 95P 5; MAX 18
```

Figure 31: Output generated from the first scenario in Figure 30



```
2021/12/03 16:45:36 completed command on worker 1 (35.227.107.253)
2021/12/03 16:45:36 preparing scenario "TLS-perf with TLS1.2"
2021/12/03 16:45:36 sending start message to 1 (35.227.107.253)
2021/12/03 16:45:36 waiting for start command on generator worker 1 (35.227.107.253)
2021/12/03 16:45:36 starting command on worker 1 (35.227.107.253)
2021/12/03 16:45:36 running command "tls-perf -T 5 --tls 1.2 -t 15 -l 5 --tickets off 35.185.111.48 443"



||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
-----------GENERATOR 35.227.107.253-----------
Running TLS benchmark with following settings:
Host:        35.185.111.48 : 443
TLS version: 1.2
Cipher:      default
TLS tickets: off
Duration:    5

( All peers are active, start to gather statistics )
TLS hs in progress 71 [3905 h/s], TCP open conns 72 [2 hs in progress], Errors 0
TLS hs in progress 72 [4032 h/s], TCP open conns 72 [3 hs in progress], Errors 0
TLS hs in progress 66 [4038 h/s], TCP open conns 68 [4 hs in progress], Errors 0
TLS hs in progress 62 [3902 h/s], TCP open conns 63 [10 hs in progress], Errors 0
TLS hs in progress 71 [4024 h/s], TCP open conns 71 [2 hs in progress], Errors 0
========================================
 TOTAL:           SECONDS 5; HANDSHAKES 19906
 HANDSHAKES/sec:  MAX 4038; AVG 3979; 95P 3902; MIN 3902
 LATENCY (ms):    MIN 3; AVG 17; 95P 28; MAX 97
2021/12/03 16:45:41 completed command on worker 1 (35.227.107.253)
```

Figure 32: Output generated from the second scenario in Figure 30

As seen by the timestamps and the red boxes provided in Figure 31 and Figure 32, both scenarios are running one after the other without the user needing to launch another instance of PTR. In the example provided above, there are only two scenarios, but if required a user can specify an arbitrary number of scenarios in a single run, allowing users to create and automate large load testing suites.

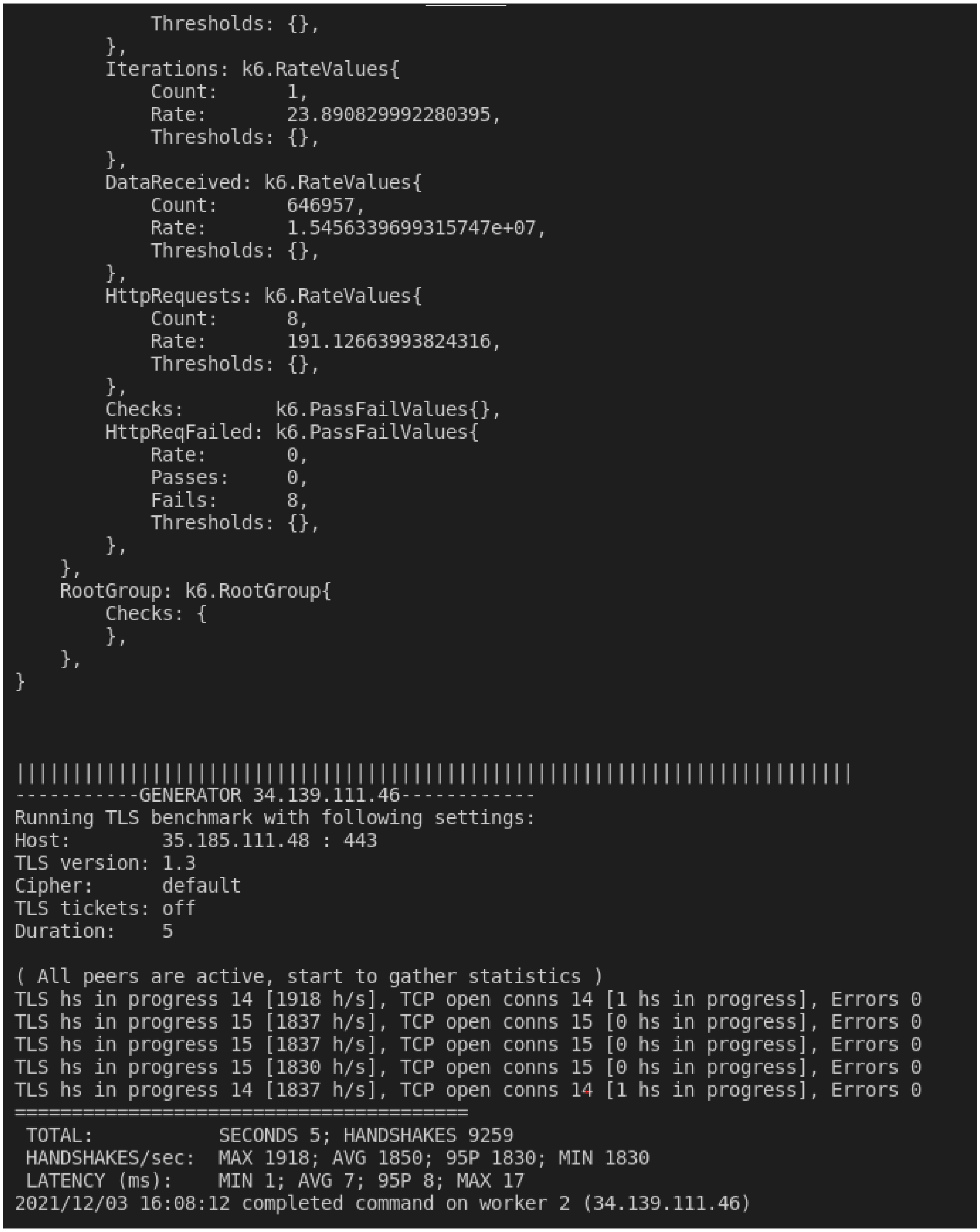
5.1.4: Running multiple generators in a single PTR execution

Oftentimes, it can be difficult to properly replicate the events that take place in a production environment. Since Fastly has a wide range of clientele, the way each client interacts with internal servers can be slightly different. For example, some customers may initiate multiple connections to Fastly's servers and issue one or two HTTP requests per connection, while others may create one connection and issue a single large request. In addition to responding to HTTP requests, servers are also responsible for handlingTLS handshakes. Using H2Load, Fastly's previous load testing tool, it was not possible to generate a sufficient number of TLS handshakes with the server. Given PTR's ability to run generators concurrently, it is possible to test new scenarios. For example, a user can run both TLS-perf and K6 to better understand how the server responds when forced to handle TLS handshakes while also processing HTTP requests. Figure 33 shows a sample command string used to run multiple generators concurrently.

```
ptr --test-config test_k6.json,test_tls_perf.json --user ubuntu --ssh-keyfile
/home/ubuntu/.ssh/id_rsa
```

Figure 33: PTR command string used to run generators concurrently.

As seen in Figure 33, to run multiple generators, the user must specify the configuration files as a comma separated string. Once this command is executed, each configuration file is loaded into a separate goroutine. Each goroutine runs its individual command string on the set of generator hosts specified in its corresponding config file. Figure 34 shows the output received by PTR once each generator's goroutine has been completed.



```
            Thresholds: {},
        },
        Iterations: k6.RateValues{
            Count:      1,
            Rate:       23.890829992280395,
            Thresholds: {},
        },
        DataReceived: k6.RateValues{
            Count:      646957,
            Rate:       1.5456339699315747e+07,
            Thresholds: {},
        },
        HttpRequests: k6.RateValues{
            Count:      8,
            Rate:       191.12663993824316,
            Thresholds: {},
        },
        Checks:        k6.PassFailValues{},
        HttpReqFailed: k6.PassFailValues{
            Rate:       0,
            Passes:     0,
            Fails:      8,
            Thresholds: {},
        },
    },
    RootGroup: k6.RootGroup{
        Checks: {
        },
    },
}



||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
-----------GENERATOR 34.139.111.46------------
Running TLS benchmark with following settings:
Host:        35.185.111.48 : 443
TLS version: 1.3
Cipher:      default
TLS tickets: off
Duration:    5

( All peers are active, start to gather statistics )
TLS hs in progress 14 [1918 h/s], TCP open conns 14 [1 hs in progress], Errors 0
TLS hs in progress 15 [1837 h/s], TCP open conns 15 [0 hs in progress], Errors 0
TLS hs in progress 15 [1837 h/s], TCP open conns 15 [0 hs in progress], Errors 0
TLS hs in progress 15 [1830 h/s], TCP open conns 15 [0 hs in progress], Errors 0
TLS hs in progress 14 [1837 h/s], TCP open conns 14 [1 hs in progress], Errors 0
========================================
 TOTAL:            SECONDS 5; HANDSHAKES 9259
 HANDSHAKES/sec:  MAX 1918; AVG 1850; 95P 1830; MIN 1830
 LATENCY (ms):    MIN 1; AVG 7; 95P 8; MAX 17
2021/12/03 16:08:12 completed command on worker 2 (34.139.111.46)
```

Figure 34: TLS-perf when run in conjunction with K6 generator (Due to K6's lengthy output, it is not possible to display its entire output in a single image.)

Running multiple generators concurrently allows Fastly to create more complex testing environments and better replicate issues seen in production.

5.2: Running PTR against Fastly infrastructure

To determine the effectiveness of our tool in a production environment, we ran PTR against Fastly's servers and monitored various metrics such as CPU usage, memory, latency, and throughput. Considering each generator type serves a different function, we used various methods of metric tracking to observe the impact that the tools had on Fastly's infrastructure. One key tool we utilized was the Grafana network dashboard, which provides detailed real-time graph panels for different types of resources, metrics, and timeframes. Using this, we could run multiple load tests, and then compare how the data from each execution appeared on the graphs over the course of the testing time frame. In the following sections, we delve further into how we used Grafana to analyze the results from running PTR on Fastly's machines.

5.2.1: Running TLS-Perf on newer-128 Core Machine vs older-72 Core Machine

One of our experiments included comparing the performance of Fastly's newer 128-core machines to their older 72-core machines when asked to respond to TLS-handshakes. Throughout running these tests, we kept all variables constant while varying only the CPU architecture of the machine we were targeting. With these results we compared any differences in CPU consumption and number of handshakes accepted, between the newer machine and the older machine. For both the newer and older machines, we ran two scenarios:

- Scenario 1: TLS-Perf with the following configurations
  - TLS version 1.2
  - Ran on Fastly's h2o production build
  - Four generators
  - 30 threads
  - Maximum of 1000 connections per thread

- Session resumption turned off

- Scenario 2: TLS-Perf with the following configurations
    - TLS version 1.3
    - Ran on Fastly's h2o production build
    - Four generators
    - 30 threads
    - Maximum of 1000 connections per thread
    - Session resumption turned off

Each scenario was run for 180 seconds with a 90 second pause in between the first and second scenarios to allow the machine resource consumption to stabilize. Setting up a complex test like this was made effortless by leveraging PTR.

The graphs shown below are from our first test against Fastly's older 72-core machine. The left bump in the graph represents the TLS-Perf test using TLS version 1.2 and the right represents the TLS-Perf test using TLS version 1.3. Figure 35 is a graph showing the CPU usage on the machine at different timestamps and Figure 36 is a graph displaying the number of TLS handshakes accepted at different timestamps.

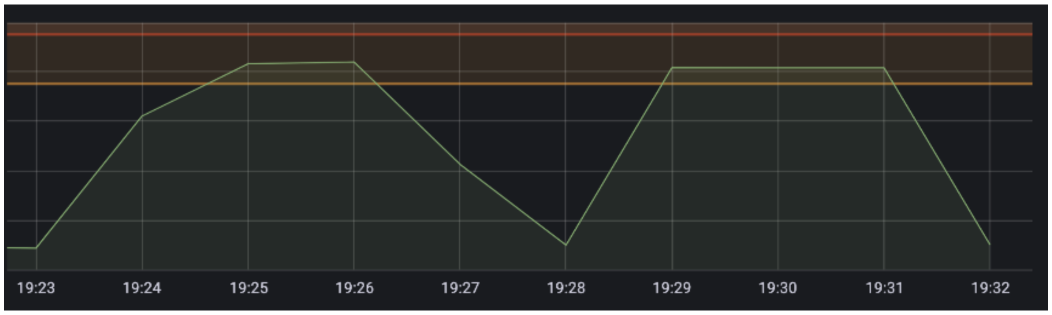


Figure 35: Grafana graph showing the CPU usage on the 72-core machine when running TLS-Perf with TLS version 1.2 (on the left) and then running TLS-Perf with TLS version 1.3 (on the right)

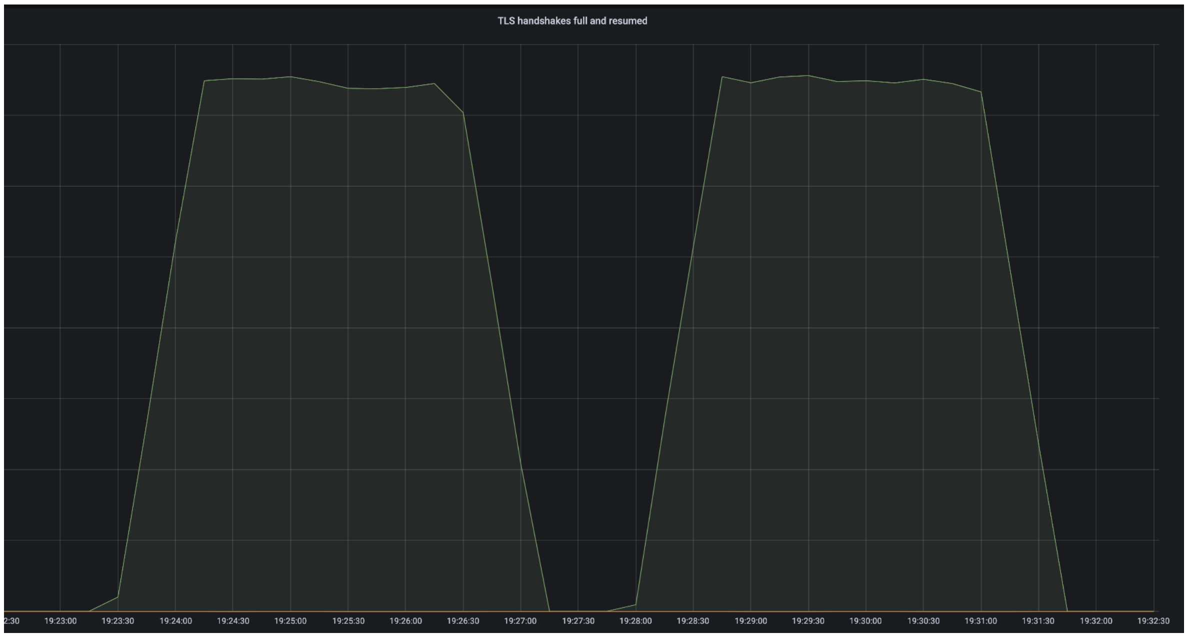


Figure 36: Result from running against the 72-core machine. Grafana graph showing the number of TLS handshakes accepted when running TLS-Perf with TLS version 1.2 (on the left) and then running TLS-Perf with TLS version 1.3 (on the right)

From the graphs, when running TLS-Perf with 30 threads and a max of 1000 connections per thread, only about 80% of the CPU is being used. The graphs show us that there is a negligible difference in CPU usage and number of TLS handshakes accepted between TLS version 1.2 and TLS version 1.3. As far as TLS handshakes, Figure 36 shows us that with this configuration, the older 72-core machine, independent of the TLS version, accepts a maximum of roughly 15,000 handshakes per second.

Figure 37 and 38 below are the CPU usage and TLS handshake graphs from our test against Fastly's newer 128-core machine. The left bump and right bump represent the TLS version 1.2 test and TLS version 1.3 test, respectively.

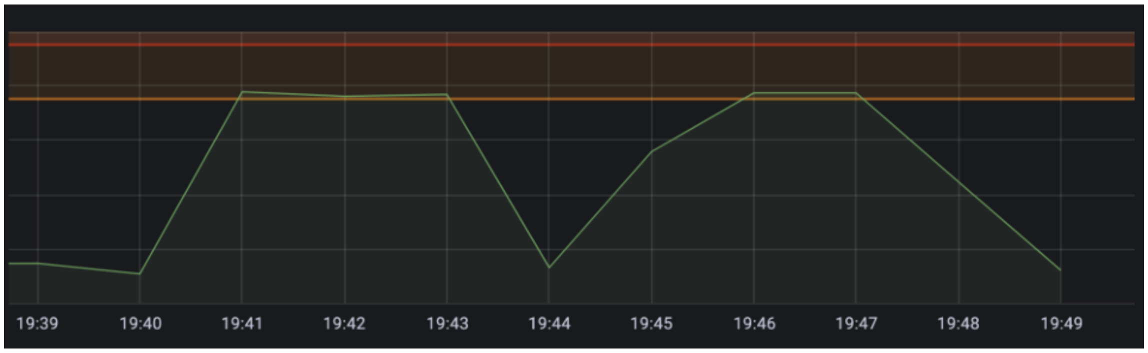


Figure 37: Grafana graph showing the CPU usage on the newer 128-core machine when running TLS-Perf with TLS version 1.2 (on the left) and then running TLS-Perf with TLS version 1.3 (on the right)

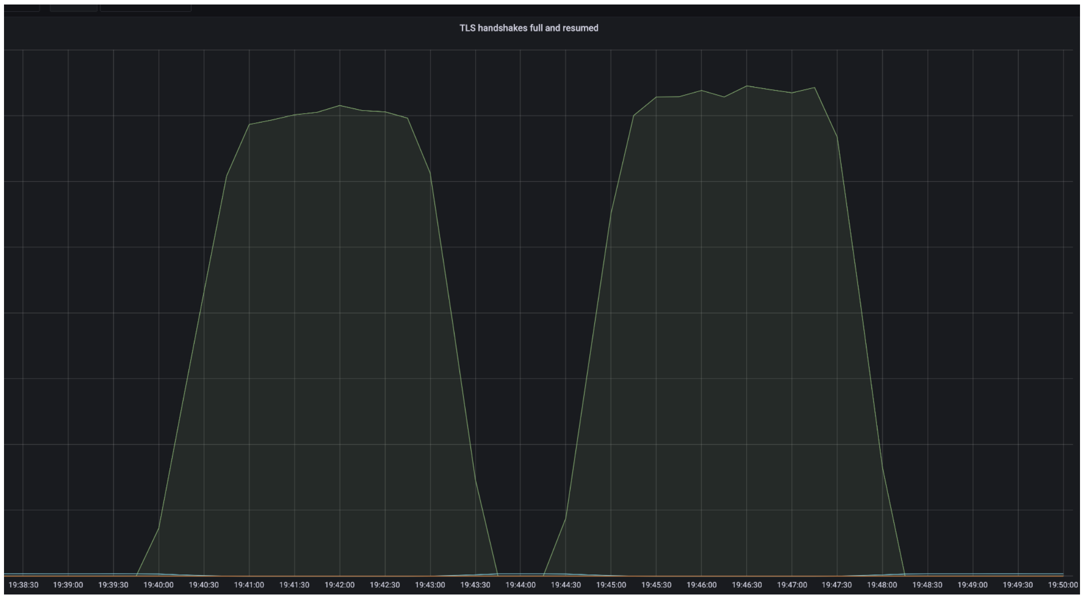


Figure 38: Result from running against the newer 128-core machine. Grafana graph showing the number of TLS handshakes accepted when running TLS-Perf with TLS version 1.2 (on the left) and then running TLS-Perf with TLS version 1.3 (on the right)

After running both tests, we observed that the newer 128-core machine is able to handle more TLS handshakes per second (about 17,500-18,000 full TLS handshakes per second) than the older 72-core machine and the newer 128-core machine used a slightly lower percentage of the CPU (a little less than 80%) compared to the 72-core machine. Looking at Figure 38, although there is a slight increase in handshakes over TLS version 1.3 compared to TLS version 1.2, there is not much of a difference. After running both tests and realizing that four generators was not maximizing the CPU's usage, we re-ran the TLS-Perf test on the newer 128-core machine comparing TLS handshakes when using four generators versus five generators. Figure 39 and 40 below show the results from that test where:

- First bump represents running TLS-Perf with four generators and TLS version 1.2
- Second bump represents running TLS-Perf with four generators and TLS version 1.3
- Third bump represents running TLS-Perf with five generators and TLS version 1.2
- Fourth bump represents running TLS-Perf with five generators and TLS version 1.3

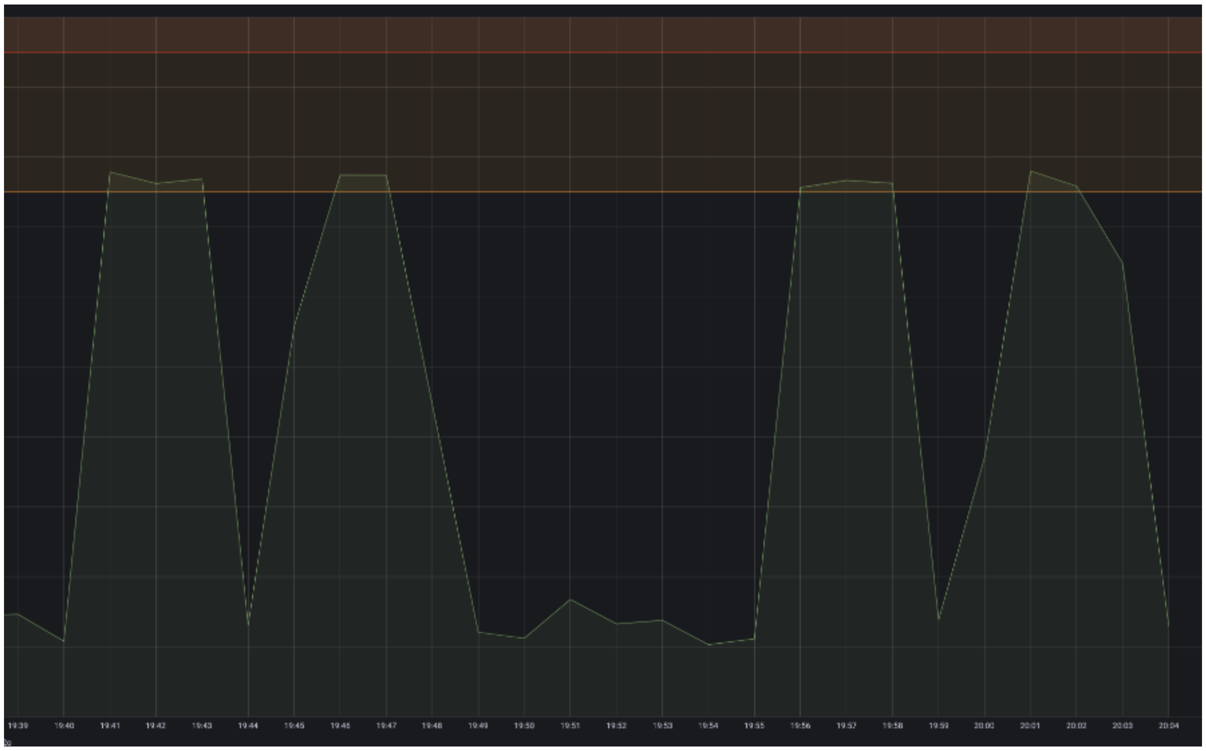


Figure 39: Grafana graph showing the CPU usage on the 128-core machine when running TLS-Perf with TLS version 1.2 vs. 1.3 and using four generators (left) and five generators (right)

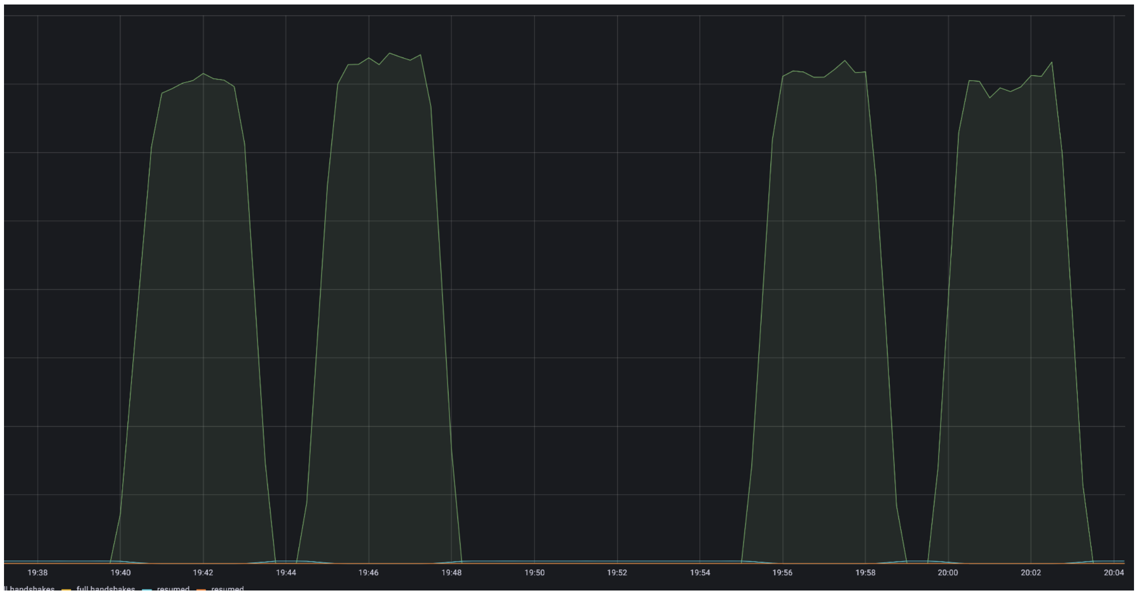


Figure 40: Result from running against the 128-core machine. Grafana graph showing the number of TLS handshakes accepted when running TLS-Perf with TLS version 1.2 vs. 1.3 and using four generators (left) and five generators (right)

When adding another generator to PTR and running the same test against the 128-core machine, we can see that CPU usage and TLS handshake rate stays the same. From these observations, a possible conclusion could be that there is no correlation between the number of generators and the TLS handshake rate accepted by the server due to a bottleneck that exists elsewhere. It is possible that the bottleneck is in the h2o (server) event/main loop shown by the graph in Figure 41 below.

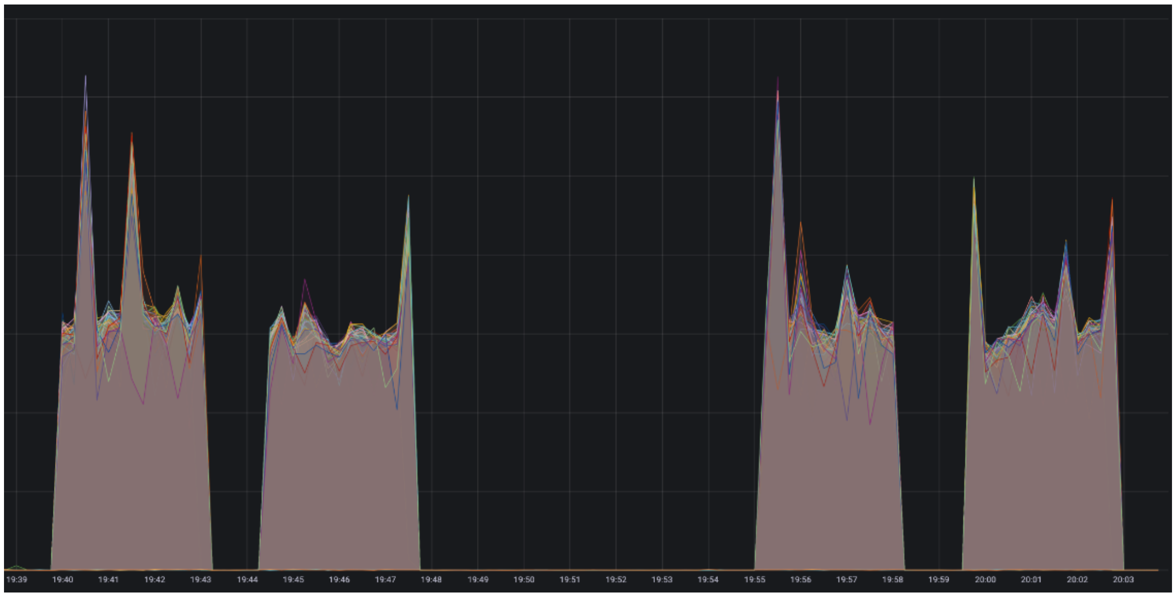


Figure 41: Grafana graph showing the latency of the h2o event loop when running the TLS-Perf test against the 128-core machine varying number of generators (four on the left, five on the right) and TLS version (first and third bump being TLS 1.2, and second and fourth being TLS 1.3)

The graph above was taken from the same test run that resulted in the graphs in Figures 39 and 40. The event loop on the server is responsible for handling the incoming connections and requests, and as the graph above shows, there seems to be a spike in latency when we issue TLS handshakes against the server. With this knowledge, we wanted to leverage PTR in order to observe throughput on the server while causing a spike in latency by issuing TLS handshakes concurrently.

5.2.2: Running K6 alone vs running TLS-Perf and K6 in parallel.

The second experiment that our team conducted was testing the performance of production machines when running K6 alone versus K6 in conjunction with TLS-Perf. For this test, our team began by configuring the target host with the production build of h2o. When K6 was run alone we specified three generator machines for the following four scenarios each separated by a 90 second pause:

Scenario 1. Requesting 10kb file for 180 seconds

Scenario 2. Requesting 100kb file for 180 seconds

Scenario 3. Requesting 1mb file for 180 seconds

Scenario 4. Requesting 10mb file for 180 seconds

Immediately following the completion of the aforementioned scenarios, our team ran another load test; however, this time K6 was run concurrently with TLS-Perf. For this portion of the experiment, we specified three unique generator hosts for both K6 and TLS-perf. The following four scenarios were run:

Scenario 1. Requesting 10kb file with TLS-Perf for 180 seconds

Scenario 2. Requesting 100kb file with TLS-Perf for 180 seconds

Scenario 3. Requesting 1mb file with TLS-Perf for 180 seconds

Scenario 4. Requesting 10mb file with TLS-Perf for 180 seconds

After reviewing the results from the experiment, our team observed that the production machines saw a drastic decrease in total throughput when asked to respond to TLS-handshakes and HTTP requests. Figure 42 shows the throughput when K6 was run alone (left side) versus when run together with TLS-perf (right side).

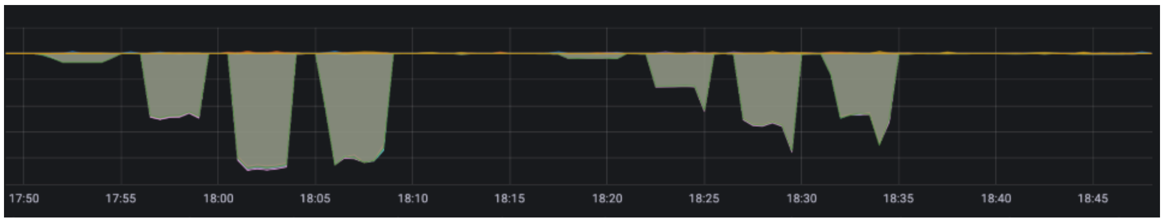


Figure 42: Total throughput measured when K6 was run alone (left side) versus K6 being run with TLS-perf (right side).

As can be seen by the graph, when K6 was run alone, there was a significant increase in throughput per file size compared to when K6 was run with TLS-Perf. The results shown in this graph are expected. When K6 is run alone, the number of handshakes that are created is low, thereby allowing the CPU to focus only on processing and responding to HTTP requests. However, if the machine is asked to both respond to handshakes and HTTP requests, the CPU spends more time completing the intensive handshake. This reduces the overall number of HTTP processed requests and correspondingly decreases the rate. One interesting observation to note about the graphs when running TLS-Perf with K6 is the spike (tooth) that appears towards the end of the test. Figure 43 highlights the tooth in the same graph seen in Figure 42 (red box).

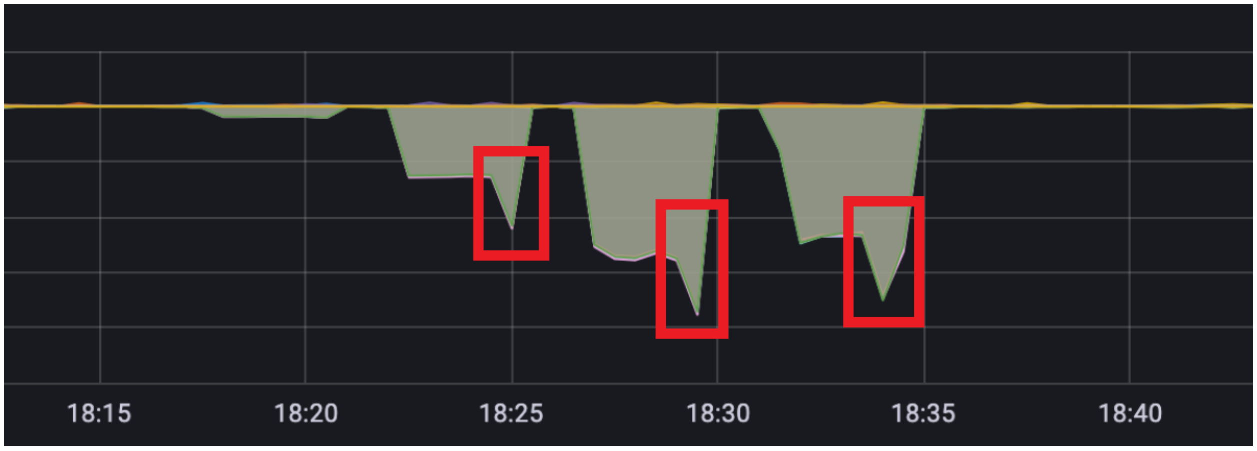


Figure 43: Graph showing the throughput spike from TLS-Perf ending and K6 continuing to issue HTTP requests.

When K6 is initialized, it requires some time to create the virtual users used to generate the load. Since TLS-Perf does not require spin up time, there is a period of time where TLS-Perf is running before K6 starts sending requests. Since both generators run for the same duration, towards the end, TLS-Perf completes its execution. Since the CPU resources are now freed from responding to handshakes, the throughput of data begins increasing, thereby causing a spike in the graph

Fastly can now test the performance of its server when asked to respond to large quantities of TLS-handshakes. Figure 44 is a graph showing the number of TLS-handshakes generated from K6 alone and in parallel with TLS-Perf.

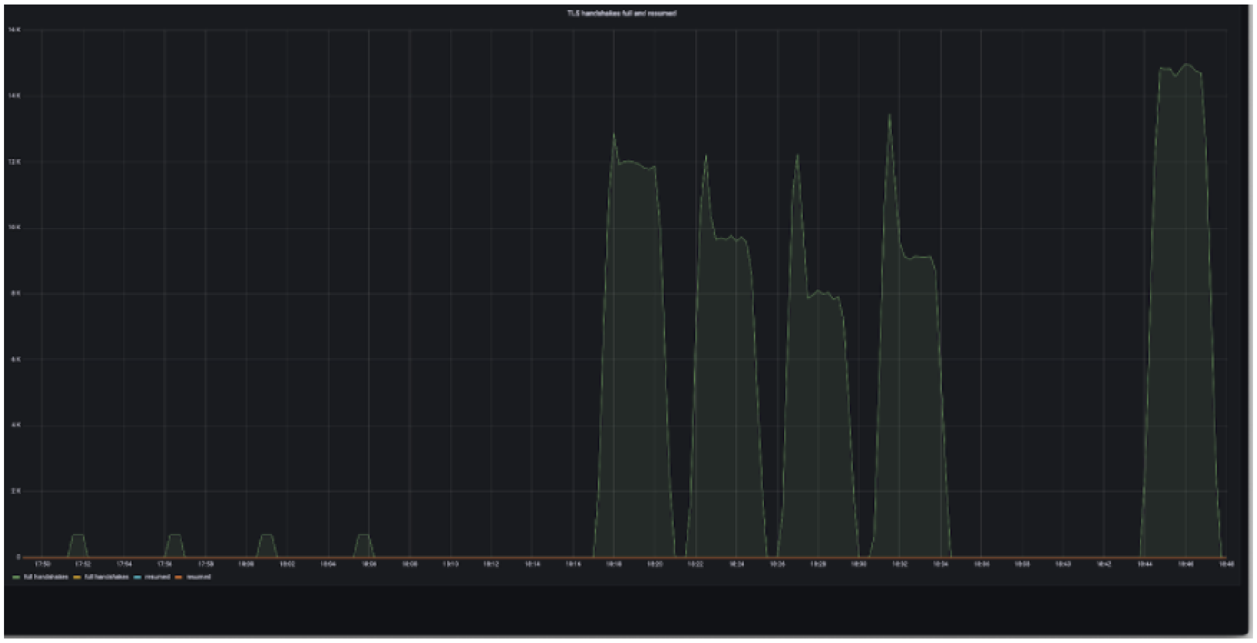


Figure 44: Total number of handshakes measured when K6 was run alone (left) versus K6 being run with TLS-perf (middle) versus TLS-perf run alone (right).

Similar to Figure 42, the left side shows the number of TLS-handshakes generated when K6 was run alone. This is an incredibly small number compared to the other peaks shown on the same graph. When K6 was run, there were on average fewer than 500 handshakes generated making it unable to accurately represent a production environment. The middle portion of the graph shows the number of handshakes generated when TLS-Perf is running with K6. When TLS-Perf is run with K6, as the object size requested increases, the number of handshakes completed decreases. The larger the file size, the more CPU cycles are spent copying data. This prevents the CPU from processing more handshakes leading to a low number of responses. An exception to note, is that the number of TLS handshakes generated when requesting ten mb files was greater than when requesting one mb files. One possible explanation for this is due to many of the ten mb HTTP requests failing. This means that the CPU was using less cycles to copy data, allowing it to process more TLS handshakes. It is also important to note that the spike in the peak now appears on the opposite side compared to the spikes shown in Figure 43. While K6 is generating virtual users to issue HTTP requests, TLS-Perf has already begun initiating handshakes with the server. For a short period of time, the server is only handling handshakes, allowing it to process them quicker - leading to a spike in the number. However, once K6 begins to issue its requests, the number of TLS handshakes decreases and eventually plateaus. Finally, on the far right side, we see the number of handshakes generated when TLS-perf is run alone. Once again, when the server is solely responding to handshakes,it is able to process more, leading to an average of 15,000 handshakes being authenticated.

Another interesting result from this experiment can be seen in the latency of the h2o event loop. Figure 45 below shows the latency of the h2o event loop. In Fastly's production build of h2o, the TLS handshake occurs within the event loop. As a result, the shape of the graph in Figure 45 resembles that of Figure 44.

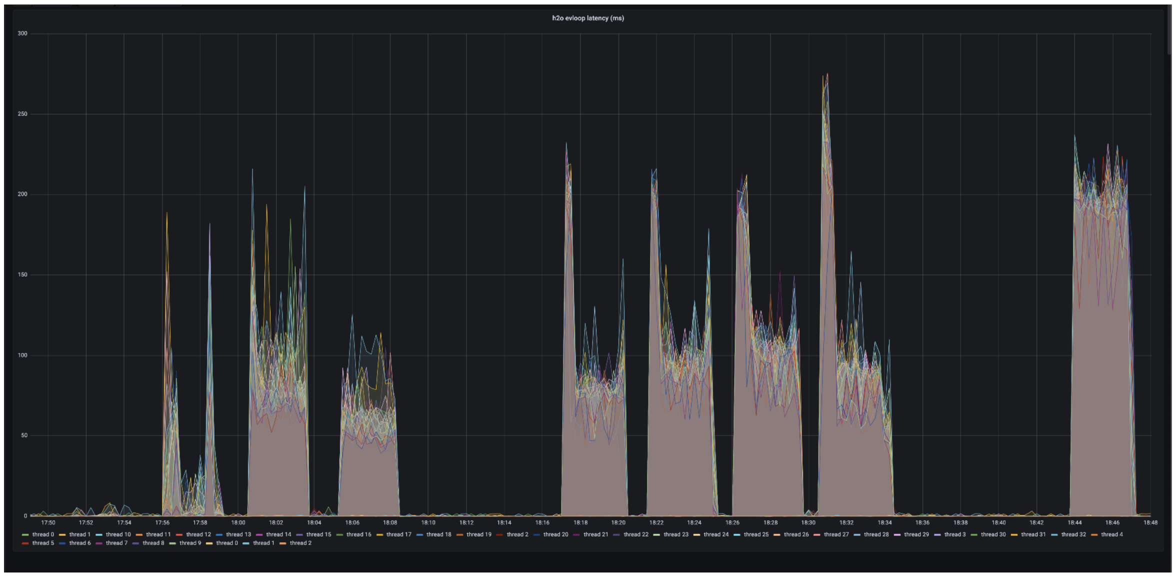


Figure 45: The event latency loop of h2o when running K6 alone versus K6 and TLS-perf versus TLS-perf alone.

As part of a future load-testing scenario, Fastly could reconfigure their h2o build by moving the portion of the code responsible for TLS-handshakes outside the event loop. After doing so, Fastly could re-run this same experiment shown here and track the performance of the server when put under TLS-handshake and HTTP request load. Moving the TLS-handshake code out of the h2o event loop could increase the overall throughput and number of HTTP requests handled, resulting in faster load times for customers.

5.2.3: Running Idle Connections alone to observe memory usage

A third notable experiment we ran against Fastly's servers was PTR with the Idle Connections program. As mentioned previously in Section 4.2.4, one of the issues Fastly faced was memory consumption on a specific build of h2o, which allocated a larger buffer size than necessary for idle TLS connections. By running the Idle Connections program, we can recreate this scenario on a production build to observe its impact on memory usage and compare the results with the unmodified h2o build.

For this experiment, we ran PTR configured to execute Idle Connections on both the production and modified builds of h2o. The test runs the program on 10 generator hosts. On each run, the program iterates through 4 NICs per host, and establishes 30,000 connections per NIC. So, in total, we should be generating 1.2 million connections against the target h2o server. We should also expect to see a spike in memory usage accompanying this, based on what was occurring in production previously. The Grafana graph below in Figure 46 shows connections over the course of the test for the two builds.

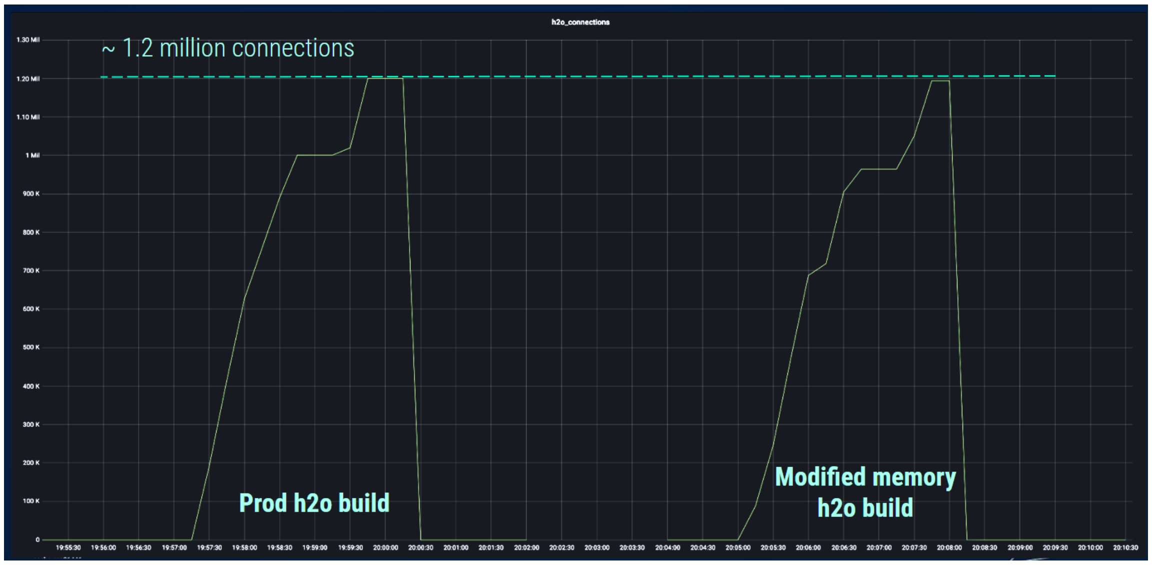


Figure 46: Total connections to both builds of h2o server on the target over a timeframe

In both builds, we see that the connections quickly increase before reaching a peak. As expected, the number of idle connections plateau right at 1.2 million. Below in Figure 47, we can observe how Residential Memory or RSS (physical RAM usage by the process) changes over the course of this same timeframe alongside connections.

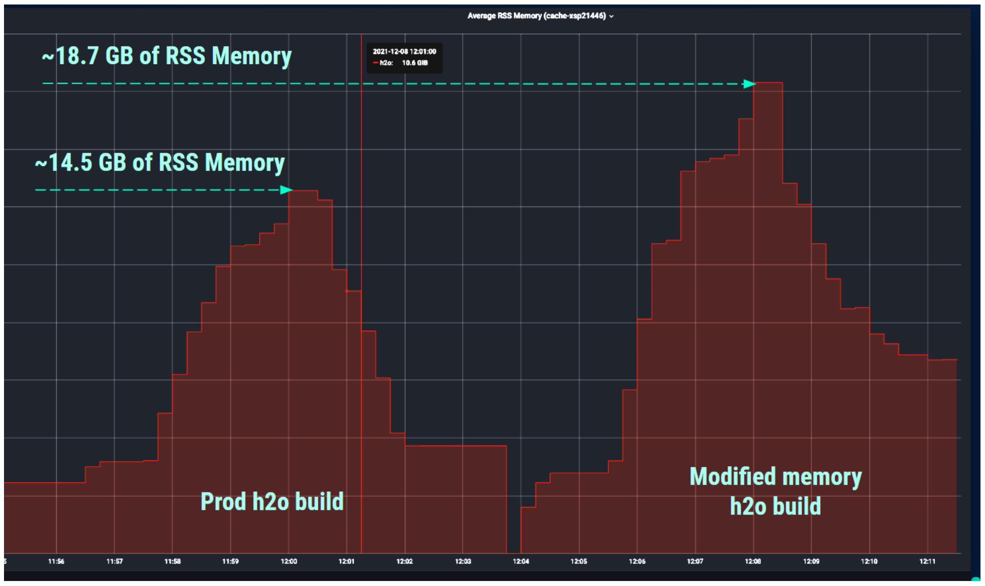


Figure 47: Average residential memory usage by both target h2o builds in GB over a timeframe

The graph shows RSS quickly increasing, corresponding to each subsequent Idle Connections program being executed within PTR. Additionally, we can see that the production h2o build plateaus at around 14.5 GB, while the modified build plateaus at about 18.7 GB. This correctly demonstrates the issue Fastly had with the new build consuming more RSS than required. Because this build initializes a buffer size of 1 MB instead of 4 KB, there is a large quantity of memory that is never read or written to. The graph below in Figure 48 shows how this translates into virtual space.

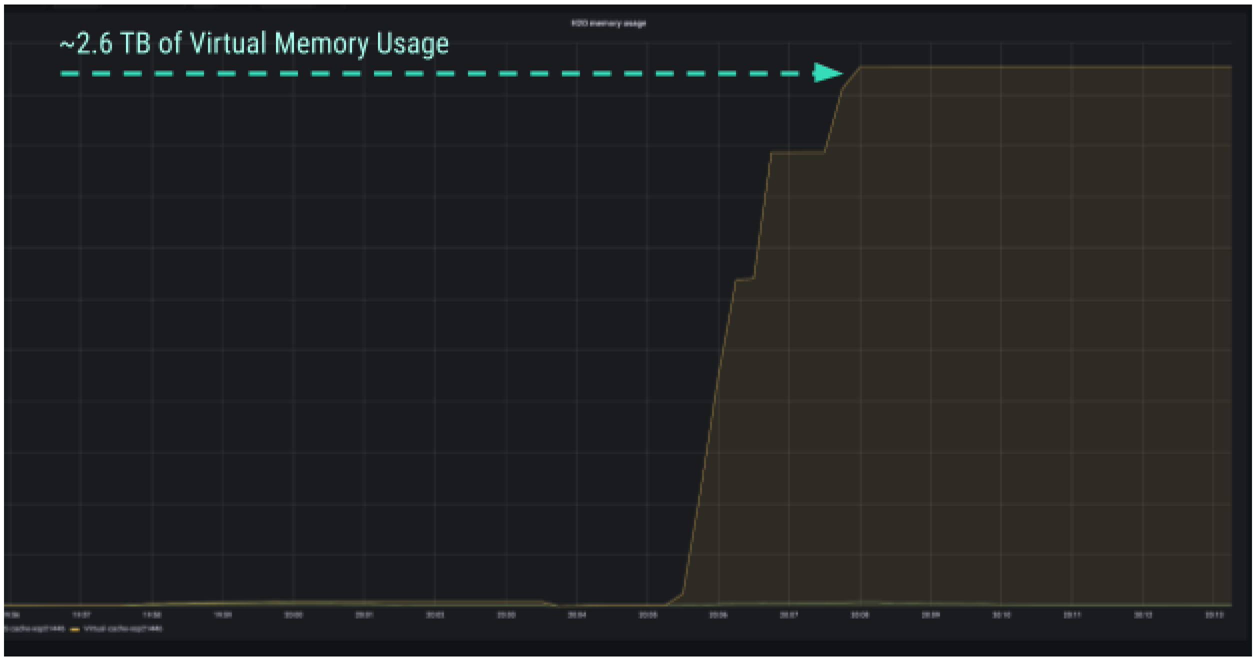


Figure 48: Memory usage for modified h2o build

This graph shows the total amount of virtual memory usage on the modified h2o build while running the experiment, which sharply inclines and peaks at nearly 2.6 terabytes. As shown, there is substantially more virtual memory that h2o allocated in comparison to the graph in Figure 47; however, because idle connections are not intensive operations, only a small amount ever gets mapped into RAM. This data is significant because we can see that the new build inefficiently allots memory that could be used better elsewhere.

Replicating the scenario that caused the original memory consumption issue through Idle Connections allows us to see exactly how much memory is being exhausted and which type is affected most heavily. Moreover, by running Idle Connections through PTR from multiple generator machines, we were able to generate a much greater load than if the program was standalone. This way, we could more accurately represent the volume of traffic that would hit Fastly's servers in a real-life scenario. Once Fastly is able to patch the receive buffer bug, they can conceivably run the same experiment again on their h2o servers and monitor the precise improvement in memory usage.

The results from these experiments provide a clear picture of how PTR strengthens Fastly's load-testing capabilities and how we can utilize its features flexibly based on the scenario at hand. In the next sections, we conclude by summarizing our findings and presenting some recommendations for improving PTR even further in the future.

CHAPTER 6: CONCLUSION

HTTP load testing is a practice used to assess the performance of servers. It primarily involves subjecting a server to simulated HTTP traffic and measuring how well that server manages under load. When running these tests, there are important metrics that can be observed such as throughput and latency. HTTP load testing is essential to ascertain if Fastly's customers may experience delays, longer wait times, and other downtime issues. For example, if Fastly updates services or partners with new companies, they may see a significant increase in server traffic and need a way to evaluate server performance beforehand. In the long-term, HTTP load tests can help Fastly ensure that they are providing reliable services to their customers.

With their current technologies, Fastly is unable to generate enough network traffic to accurately represent a production environment and, as a result, they cannot precisely measure server performance. In particular, Fastly's load test technology does not support the ability to run parallel tests of different load types and sizes. With our project, we addressed these limitations by discovering the open-source tool K6 and extending the wrapper program Performance Test Runner (PTR). When comparing K6 to Fastly's criteria, we found some limitations in the K6 tool. One limitation includes the inability to generate a large amount of TLS handshakes; to fix this, we leveraged an open-source tool, called TLS-Perf, within PTR. Additionally, K6 lacked support for generating idle connections to an endpoint. To address this, we developed a program that was able to produce a large number of idle connections. With these extensions integrated into PTR, we reproduced specific production scenarios related to TLS handshakes and idle connections.

After testing our developments against Fastly's infrastructure, PTR revealed potential bottlenecks and server inefficiencies. PTR allowed us to compare the performance between Fastly's older machines and their newer machines. By using TLS-Perf we were able to generate CPU intensive operations to analyze which machines could handle more TLS handshakes. While running an auxiliary test, we discovered that after a certain amount of load, the newer machine could not process any more TLS handshakes. This led to the discovery of a potential bottleneck in their server.

Another inefficiency includes decreased server throughput when targeted with both TLS handshakes and HTTP requests. With the help of PTR's ability to leverage multiple tools, we were able to visualize the change in server throughput when subjected to handshakes and HTTP requests. By fixing this server-side inefficiency, Fastly may see a higher throughput of HTTP requests even when a large number of TLS handshakes are handled.

Finally, we were able to replicate the memory allocation problem that had been occurring in a modified h2o build and compare metrics like RSS and virtual space to those of the production version. Our findings from the experiment confirmed that the new build consumed unnecessary memory as opposed to the unmodified build. From there, we identified precisely how much of a difference there was. Given this information, Fastly can better pinpoint the issue in the modified h2o build if they choose to patch it for more efficient memory usage in a future release.

CHAPTER 7: FUTURE WORK

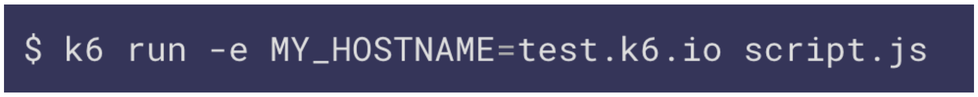
While we were able to fulfill a majority of Fastly's requirements, there are still steps that can be taken to improve their load-testing flow in the future. Below we discuss some of our recommendations.

One of our recommendations is to modify the design of PTR such that load-testing using PTR includes cleaning up any files that get copied over to generators. This would avoid cluttering the generator machines with K6 JavaScript and JSON files. Initially, we thought it would make sense to add a cleanup function as a part of the generator interface such that each generator would remove unnecessary files at the end of execution. However, only K6 would need a cleanup function because it is the only tool that needs files copied over to the generator hosts. By adding a cleanup function to the generator interface we would be breaking that interface segregation design principle that states no code should be forced to depend on or implement methods it does not need.

Another feature that would be beneficial while testing PTR is to allow the user to specify an "ulimit" field inside of a PTR configuration file. By adding this feature, the user would be able to instruct PTR to modify the command string such that when executed, the generator hosts increase the number of available file descriptors before running a tool. For example, in order to generate a large number of idle connections using the Idle Connections program, the generator hosts would need to have enough file descriptors available to support all the connections.

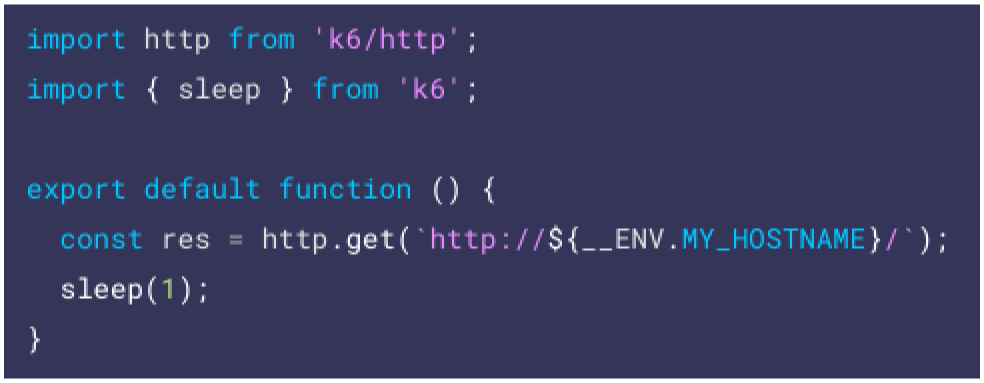
We also recommend that Fastly extend the "bind to NICs" functionality to all of PTR. For context, one important feature we integrated was the ability to iterate through each NIC in the Idle Connections program. During testing, we recognized that this capability would be a useful addition for the other tools wrapped in PTR as well, because it allowed us to generate several times more load from one generator machine and use fewer hosts in general.

An issue we ran into while testing was having to specify the target host twice, once in the K6 JavaScript configuration file within the "batch" option, and once in the PTR configuration file. This is inconvenient for two reasons: 1) the program becomes slightly redundant, and 2) if the user wanted to make a change to the target host, they would have to remember to fix it in two different locations. Therefore, to simplify this process, we suggest that Fastly modify the PTR configuration file to allow the user to specify the target host along with other K6 options. From here, these options can be passed into the command string as environment variables that can be read in by a K6 JavaScript configuration file. Figure 49 and 50 below shows how an environment variable, like the target host, would be read in when running K6.



```
$ k6 run -e MY_HOSTNAME=test.k6.io script.js
```

Figure 49: Example K6 command string that specifies a hostname environment variable (K6.io, n.d)

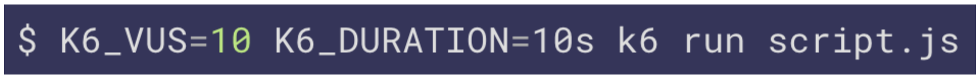


```
import http from 'k6/http';
import { sleep } from 'k6';

export default function () {
  const res = http.get(`http://${__ENV.MY_HOSTNAME}/`);
  sleep(1);
}
```

Figure 50: Example K6 JavaScript configuration file that reads in an environment variable named “MY_HOSTNAME” (K6.io, n.d)

The figures above give an example of how an environment variable for hostname can be specified on the command line and then read in from a “template” K6 JavaScript configuration file. K6 also allows the user to specify predefined environment variables for most of the options that exist in the options object as seen in Figure 51 below.



```
$ K6_VUS=10 K6_DURATION=10s k6 run script.js
```

Figure 51: Example K6 command string that specifies the number of virtual users and duration on the command line instead of in the JavaScript configuration file (K6.io, n.d)

One caveat for Fastly's future development of K6 is the project's upcoming HTTP API implementation update. In our process of attempting to merge the force HTTP1 feature into K6 (https://github.com/grafana/k6/pull/2222), the creators disclosed that the project would soon be undergoing a major update to its HTTP utility. With this, we recommend that Fastly monitor changes made to the K6 project, because the way to enable the force HTTP1 functionality may change.

As technologies continue to change, Fastly's infrastructure will need to adapt, giving rise to potential issues observed in production. By integrating new tools into PTR, Fastly will be better prepared to evaluate their server's performance. This will allow them to diversify their testing capabilities to cover more specific areas of their system.

REFERENCES

A typical HTTP session. HTTP | MDN. (2021, August 31). Retrieved October 3, 2021, from https://developer.mozilla.org/en-US/docs/Web/HTTP/Session.

Biketi, B. (2021, July 5). *Comparison between the HTTP/3 and HTTP/2 protocols.* Section. Retrieved September 28, 2021, from https://www.section.io/engineering-education/http3-vs-http2/.

Boucheron, B. (2021, January 14). *An introduction to load testing.* DigitalOcean. Retrieved October 6, 2021, from https://www.digitalocean.com/community/tutorials/an-introduction-to-load-testing#load-testing-basics.

Diagram showing HTTP process. (n.d.). Study CCNA. Retrieved October 3, 2021, from https://study-ccna.com/http-https/.

Gudeliauskas, D. (2021, May 15). *What is SSL/TLS and HTTPS? The importance of a secure web explained.* Hostinger Tutorials. Retrieved September 28, 2021, from https://www.hostinger.com/tutorials/what-is-ssl-tls-https.

Grigorik, I. Surma. *Introduction to HTTP/2*. (n.d.). Google. Retrieved October 3, 2021, from https://developers.google.com/web/fundamentals/performance/http2.

Hamilton, T. (2021, August 28). *Load testing tutorial: what is? how to? (with examples).* Guru99. Retrieved October 6, 2021, from https://www.guru99.com/load-testing-tutorial.html

HTTP vs. HTTPS and what a hacker sees. (2019). Botify. Retrieved October 3, 2021, from https://www.botify.com/blog/what-is-a-secure-website-https-vs-http

Hypertext protocol (HTTP): client-server communication. CUNY Manifold App . (n.d.). Retrieved October 3, 2021, from https://cuny.manifoldapp.org/read/hypertext-protocol-http/section/c3bfe85d-46f0-44ba-add2-75ac3eaf2cb3.

K6.io. (n.d.). *Example K6 command string that specifies a hostname environment variable.* Environment variables. Retrieved December 4, 2021, from https://k6.io/docs/using-k6/environment-variables/.

K6.io. (n.d.). *Example K6 JavaScript configuration file that reads in an environment variable named "MY_HOSTNAME".* Environment variables. Retrieved December 4, 2021, from https://k6.io/docs/using-k6/environment-variables/.

K6.io. (n.d.). *Example K6 command string that specifies the number of virtual users and duration on the command line instead of in the JavaScript configuration file. Environment variables*. Retrieved December 4, 2021, from https://k6.io/docs/using-k6/environment-variables/.

Krizhanovsky, A., Koveshnikov, I., & Alexander. (2020). *Tempesta-tech/TLS-Perf: TLS handshakes benchmarking tool*. GitHub. Retrieved December 1, 2021, from https://github.com/tempesta-tech/tls-perf.

Load testing metrics explained. LoadStorm. (2017, March 10). Retrieved October 7, 2021, from https://loadstorm.com/load-testing-metrics/.

MDN. (2021, October 3). *An overview of HTTP.* Developer Mozilla. Retrieved September 28, 2021, from https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview.

MDN. (2021, October 4). *HTTP headers*. Developer Mozilla. Retrieved September 28, 2021, from https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers.

Menascé, D. A. (2002). *Load testing of web sites*. IEEE Internet computing, 6(4), 70-74.

Metrics. K6. (n.d.). Retrieved December 4, 2021, from https://k6.io/docs/using-k6/metrics/.

Monaghan, M. (2021, October 5). *Website load time statistics: why speed matters in 2021.* Website Builder Expert. Retrieved October 6, 2021, from https://www.websitebuilderexpert.com/building-websites/website-load-time-statistics/

Remote access via bastion host. (n.d.). Goteleport. Retrieved 2021, from https://goteleport.com/blog/images/2021/bastion/remote-access-via-bastion.png.

Test life cycle. K6. (n.d.). Retrieved December 3, 2021, from https://k6.io/docs/using-k6/test-life-cycle/.

Tsujikawa, T. (2015, February 16). *Nghttp2: HTTP/2 C library.* Nghttp2. Retrieved October 4, 2021, from https://nghttp2.org/.

Tsujikawa, T. (2016). *H2Load - http/2 benchmarking tool*. nghttp2. Retrieved October 4, 2021, from https://nghttp2.org/documentation/H2Load-howto.html.

What happens in a TLS handshake? Cloudflare. (n.d.). Retrieved October 4, 2021, from https://www.cloudflare.com/learning/ssl/what-happens-in-a-tls-handshake/.

What is HTTPS? Cloudflare. (n.d.). Retrieved October 4, 2021, from https://www.cloudflare.com/learning/ssl/what-is-https/.

What is latency? Cloudflare. (n.d.). Retrieved October 3, 2021, from https://www.cloudflare.com/learning/performance/glossary/what-is-latency/.

APPENDIX A: METRICS OFFERED BY K6

Metric Name	Type	Description
“http_reqs”	Counter	HTTP requests generated, in total
“iterations”	Counter	Total number of times the virtual users have executed the JS script
“dropped_iterations”	Counter	Total number of iterations not started due to insufficient number of virtual users or amount of time
“data_received”	Counter	The amount of data received
“data_sent”	Counter	The amount of data sent
“http_req_blocked”	Trend	Time spent blocked before issuing the request (while waiting for a free TCP connection slot)
“http_req_connecting”	Trend	Time spent establishing connection to the host
“http_req_tls_handshaking”	Trend	Time spent TLS handshaking with the host
“http_req_sending”	Trend	Time spent sending data to the host
“http_req_waiting”	Trend	Time spent waiting for response from the host, this is typically known as “time to first byte” or TTFB
“http_req_receiving”	Trend	Time spent receiving data from the host
“http_req_duration”	Trend	Total time for the request. The sum of sending time, waiting time, and receiving time
“iteration_duration”	Trend	The amount of time it took to execute one full iteration of the default function

“http_req_failed”	Rate	Rate of failed requests
“checks”	Rate	Rate of successful checks
“vus”	Gauge	The number of active virtual users
“vus_max”	Gauge	The max possible number of virtual users

Appendix B: PTR Output when running a K6 scenario

```
sjordhani@ubuntu:~/Desktop/ptr-mqp$ ptr --test-config test2_k6.json --user ubuntu
--ssh-keyfile /home/sjordhani/.ssh/id_rsa
2021/12/04 09:02:32 reading config from: test2_k6.json
2021/12/04 09:02:32 SSH agent socket exists, will use agent for authentication
2021/12/04 09:02:32 configuring fallback keyboard interactive authentication
2021/12/04 09:02:32 establishing connection to target host: 35.185.111.48
2021/12/04 09:02:33 establishing connection to generator host: 35.227.107.253
2021/12/04 09:02:34 establishing connection to generator host: 34.139.111.46
-----------------------------------------------------------------------------------
---------
-------------------Generator: 35.227.107.253:22 (generator-index:
0)------------------------
-----------------------------------------------------------------------------------
---------
2021/12/04 09:02:35 SSH agent socket exists, will use agent for authentication
2021/12/04 09:02:35 configuring fallback keyboard interactive authentication
bastion was not detected -- using direct connect instead
attempting to copy file from
/home/sjordhani/Desktop/ptr-mqp/internal/generators/k6/k6_js/test.js to
/home/ubuntu/remote_jsfile.js (file-index: 0)
file was shared successfully
-----------------------------------------------------------------------------------
---------
-------------------Generator: 34.139.111.46:22 (generator-index:
1)------------------------
-----------------------------------------------------------------------------------
---------
2021/12/04 09:02:36 SSH agent socket exists, will use agent for authentication
2021/12/04 09:02:36 configuring fallback keyboard interactive authentication
bastion was not detected -- using direct connect instead
attempting to copy file from
/home/sjordhani/Desktop/ptr-mqp/internal/generators/k6/k6_js/test.js to
/home/ubuntu/remote_jsfile.js (file-index: 0)
file was shared successfully
-----------------------------------------------------------------------------------
---------
2021/12/04 09:02:37 preparing scenario "test to see if PTR works"
2021/12/04 09:02:37 sending start message to 1 (35.227.107.253)
2021/12/04 09:02:37 sending start message to 2 (34.139.111.46)
2021/12/04 09:02:37 waiting for start command on generator worker 2 (34.139.111.46)
2021/12/04 09:02:37 starting command on worker 2 (34.139.111.46)
2021/12/04 09:02:37 waiting for start command on generator worker 1
(35.227.107.253)
2021/12/04 09:02:37 starting command on worker 1 (35.227.107.253)
2021/12/04 09:02:37 running command "k6 run -q --summary-export /tmp/file.json
/home/ubuntu/remote_jsfile.js > /dev/null 2>&1; cat /tmp/file.json;"
2021/12/04 09:02:37 running command "k6 run -q --summary-export /tmp/file.json
/home/ubuntu/remote_jsfile.js > /dev/null 2>&1; cat /tmp/file.json;"
```

```
2021/12/04 09:02:38 completed command on worker 2 (34.139.111.46)
2021/12/04 09:02:38 completed command on worker 1 (35.227.107.253)
----------------------- Generator 34.139.111.46 -----------------------
RESULTS: k6.Stats{
    StatsMetrics: k6.Metrics{
        HostName:       "34.139.111.46",
        HttpReqSending: k6.TimeValues{
            Avg:        0.01786925,
            Min:        0.006441,
            Med:        0.01085,
            Max:        0.043673,
            P90:        0.034449799999999996,
            P95:        0.03906139999999999,
            Thresholds: {},
        },
        HttpReqBlocked: k6.TimeValues{
            Avg:        1.037470375,
            Min:        0.000228,
            Med:        0.0003095,
            Max:        8.297432,
            P90:        2.4895879999999986,
            P95:        5.393509999999996,
            Thresholds: {},
        },
        HttpReqDuration: k6.TimeValues{
            Avg:        7.246578125000001,
            Min:        1.74959,
            Med:        8.7553105,
            Max:        9.579435,
            P90:        9.500209,
            P95:        9.539822000000001,
            Thresholds: {},
        },
        HttpReqWaiting: k6.TimeValues{
            Avg:        6.22227425,
            Min:        1.500684,
            Med:        7.839251,
            Max:        7.931677,
            P90:        7.9171891,
            P95:        7.92443305,
            Thresholds: {},
        },
        HttpReqReceiving: k6.TimeValues{
            Avg:        1.006434625,
            Min:        0.218409,
            Med:        0.9233245,
            Max:        1.660729,
            P90:        1.5975883,
            P95:        1.62915865,
            Thresholds: {},
        },
        HttpReqTlsHandshaking: k6.TimeValues{
            Avg:        0.443141875,
        Min:        0,
        Med:        0,
        Max:        3.545135,
        P90:        1.0635404999999993,
        P95:        2.3043377499999984,
        Thresholds: {},
    },
    HttpReqDurationExpectedTrue: k6.TimeValues{
        Avg:        7.246578125000001,
        Min:        1.74959,
        Med:        8.7553105,
        Max:        9.579435,
        P90:        9.500209,
        P95:        9.539822000000001,
        Thresholds: {},
    },
    HttpReqConnecting: k6.TimeValues{
        Avg:        0.5808485,
        Min:        0,
        Med:        0,
        Max:        4.646788,
        P90:        1.3940363999999992,
        P95:        3.0204121999999973,
        Thresholds: {},
    },
    IterationDuration: k6.TimeValues{
        Avg:        20.092399,
        Min:        20.092399,
        Med:        20.092399,
        Max:        20.092399,
        P90:        20.092399,
        P95:        20.092399,
        Thresholds: {},
    },
    DataSent: k6.RateValues{
        Count:      4215,
        Rate:       197109.48180782364,
        Thresholds: {},
    },
    Iterations: k6.RateValues{
        Count:      1,
        Rate:       46.763815375521624,
        Thresholds: {},
    },
    DataReceived: k6.RateValues{
        Count:      647133,
        Rate:       3.0262408135407433e+07,
        Thresholds: {},
    },
    HttpRequests: k6.RateValues{
        Count:      8,
        Rate:       374.110523004173,
        Thresholds: {},
        },
        Checks:        k6.PassFailValues{},
        HttpReqFailed: k6.PassFailValues{
            Rate:       0,
            Passes:     0,
            Fails:      8,
            Thresholds: {},
        },
    },
    RootGroup: k6.RootGroup{
        Checks: {
        },
    },
}
----------------------- Generator 35.227.107.253 -----------------------
RESULTS: k6.Stats{
    StatsMetrics: k6.Metrics{
        HostName:       "35.227.107.253",
        HttpReqSending: k6.TimeValues{
            Avg:        0.012639999999999998,
            Min:        0.006338,
            Med:        0.0095685,
            Max:        0.030479,
            P90:        0.022409399999999996,
            P95:        0.026444199999999994,
            Thresholds: {},
        },
        HttpReqBlocked: k6.TimeValues{
            Avg:        0.5501717500000001,
            Min:        0.000151,
            Med:        0.00017700000000000002,
            Max:        4.399912,
            P90:        1.3202367999999989,
            P95:        2.8600743999999976,
            Thresholds: {},
        },
        HttpReqDuration: k6.TimeValues{
            Avg:        4.479921875,
            Min:        2.254774,
            Med:        4.842086,
            Max:        5.846828,
            P90:        5.8211366,
            P95:        5.833982300000001,
            Thresholds: {},
        },
        HttpReqWaiting: k6.TimeValues{
            Avg:        3.4693914999999995,
            Min:        1.690395,
            Med:        4.0850475,
            Max:        4.24742,
            P90:        4.2463336,
            P95:        4.2468768,
            Thresholds: {},
        },
        HttpReqReceiving: k6.TimeValues{
            Avg:        0.9978903750000001,
            Min:        0.078114,
            Med:        1.1250625,
            Max:        1.590537,
            P90:        1.5672452000000001,
            P95:        1.5788911,
            Thresholds: {},
        },
        HttpReqTlsHandshaking: k6.TimeValues{
            Avg:        0.2542875,
            Min:        0,
            Med:        0,
            Max:        2.0343,
            P90:        0.6102899999999997,
            P95:        1.3222949999999989,
            Thresholds: {},
        },
        HttpReqDurationExpectedTrue: k6.TimeValues{
            Avg:        4.479921875,
            Min:        2.254774,
            Med:        4.842086,
            Max:        5.846828,
            P90:        5.8211366,
            P95:        5.833982300000001,
            Thresholds: {},
        },
        HttpReqConnecting: k6.TimeValues{
            Avg:        0.28017,
            Min:        0,
            Med:        0,
            Max:        2.24136,
            P90:        0.6724079999999996,
            P95:        1.4568839999999987,
            Thresholds: {},
        },
        IterationDuration: k6.TimeValues{
            Avg:        18.664832,
            Min:        18.664832,
            Med:        18.664832,
            Max:        18.664832,
            P90:        18.664832,
            P95:        18.664832,
            Thresholds: {},
        },
        DataSent: k6.RateValues{
            Count:      4005,
            Rate:       193210.09086373882,
            Thresholds: {},
        },
        Iterations: k6.RateValues{
            Count:      1,
            Rate:       48.24221994100845,
            Thresholds: {},
        },
        DataReceived: k6.RateValues{
            Count:      647133,
            Rate:       3.121913251708462e+07,
            Thresholds: {},
        },
        HttpRequests: k6.RateValues{
            Count:      8,
            Rate:       385.9377595280676,
            Thresholds: {},
        },
        Checks:        k6.PassFailValues{},
        HttpReqFailed: k6.PassFailValues{
            Rate:       0,
            Passes:     0,
            Fails:      8,
            Thresholds: {},
        },
    },
    RootGroup: k6.RootGroup{
        Checks: {
        },
    },
}
```