

TABLE OF CONTENT

TABLE OF CONTENT	1
PERFORMANCE TESTING	2
Types of Performance Testing	2
Load Testing:	2
Stress Testing:	2
Endurance Testing:	2
Spike Testing:	2
Volume Testing:	3
Scalability Testing:	3
Performance Testing Life Cycle	3
Performance Testing Tools	5
GATLING	6
Why Gatling	7
Scala	7
Maven	8
Gatling Core Package	8
Gatling HTTP Package	8
HTTP Headers	9
THROUGHPUT	9
PERFORMANCE TUNING	10
BENCHMARKING	11
PERFORMANCE ENGINEERING	11
PERFORMANCE ENGINEERING vs SRE	12
ASSERTION	12
Applicable to response time	13
Applicable to number of requests (all, failed or successful)	13
SESSION	14
SCENARIO	15
INJECTIONS	15
Open Workload Models	15
Closed Workload Models	19
PERCENTILE	20
INFLUXDB & GRAFANA WITH GATLING	21

PERFORMANCE TESTING

Performance testing is a form of non-functional software testing used to measure the performance of a system, application, or website under specific conditions, such as a high volume of concurrent users or a heavy workload. The goal of performance testing is to identify performance bottlenecks, measure system response times, and determine whether the system meets its performance requirements under different scenarios. Performance testing typically involves generating load on the system under test by simulating realistic user behavior and interactions. This can be done using a variety of tools and techniques, such as load testing tools, performance monitoring tools, and manual testing methods.

Types of Performance Testing

Load Testing:

This type of testing involves putting the system under varying loads to see how it performs. The goal is to identify performance bottlenecks and determine the maximum capacity of the system.

Stress Testing:

This type of testing involves putting the system under extreme conditions, such as heavy user loads or limited resources, to see how it behaves. The goal is to identify how the system responds under pressure and to determine its breaking point.

Endurance Testing:

This type of testing involves running the system under a sustained load over a period of time to see how it performs over time. The goal is to identify how the system performs over an extended period of time and to detect any performance degradation over time.

Spike Testing:

This type of testing involves rapidly increasing the load on the system to see how it responds. The goal is to identify how the system performs under sudden, unexpected spikes in user traffic.

Volume Testing:

This type of testing involves testing the system under a large volume of data to see how it performs. The goal is to identify how the system responds to large amounts of data and to determine whether it can handle the required volumes of data.

Scalability Testing:

This type of testing involves testing the system's ability to handle increasing loads as the system grows. The goal is to identify the system's scalability limits and to determine how well it can handle increased loads as the system produces.



Performance Testing Life Cycle

There are mainly 6 steps in performance testing. They are:

1. Requirements Analysis

It is one of the most important and critical steps to understand the non-functional requirements in PTLC. It helps to evaluate the degree of compliance with non-functional needs.

2. Test Planning

The second defines how to approach Performance Testing for the identified critical scenarios. You need to address the kind of performance testing and the tools required.

3. Test Design

This phase involves script generation using the identified testing tool in a dedicated environment. The script enhancements are needed to be done and unit tested.

4. Test Execution

The next phase is dedicated to the test engineers who design scenarios based on identified workload and load the system with concurrent virtual users.

5. Result Analysis

In this phase, experienced test engineers analyze and review the collected log files. Tuning recommendations will be given if any conflicts are identified.

6. Result Analyze, Report, and Retest

This is the last phase in PTLC which involves benchmarking and providing a recommendation to the client.

Performance Testing Tools

There are great tools available for performance testing in the market. The purpose of conducting performance testing is to establish how efficiently the system runs under different workloads. Automated testing tools help conduct the test most effectively. Each tool is different in its capability and scope. These tools test the applications on various factors like performance testing, GUI testing, functional testing, etc. Tools that are in demand, easy to learn, and effective for the testing requirement are recommended for use.

Commercial tools are typically paid and provide comprehensive performance testing solutions with advanced features and support. These tools are often used in large enterprises where performance testing is critical and requires high-quality tools and support.

Open-source tools are free to use and have a community of developers contributing to the development and maintenance of the tool. Open-source tools can be a good option for small to medium-sized businesses that do not have the budget for commercial tools or do not require the advanced features provided by commercial tools.





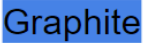


Open-source tools like **Gatling** can be more flexible in terms of customization and integration with other tools and systems. This can be important if you have specific requirements or want to build a customized testing solution. Gatling is a powerful and flexible open-source tool that can be a good choice for organizations looking to conduct performance testing without the high cost associated with commercial tools.

Performance testing is carried out with various Trimble teams

Gatling is an open-source tool alternative for Very Expensive Web Load tools.

- On an average 50,000\$ is being charged by radview for just 400 Virtual users.
- 8000\$ paid every year for webload support.

Based on customer applications and various scenarios we are using more internal, open-source, and paid load testing, monitoring, and performance tuning tools.

Paid Performance Tools	Free Performance Tool	JVM Profilers	Internal Tools
 	  	 	<ul style="list-style-type: none"> • PERAGENT • HDaS • Packet Simulator • Upload File Generator • DataFile Generator • Other Perl, Java and Shell Utilities for Log Analysis

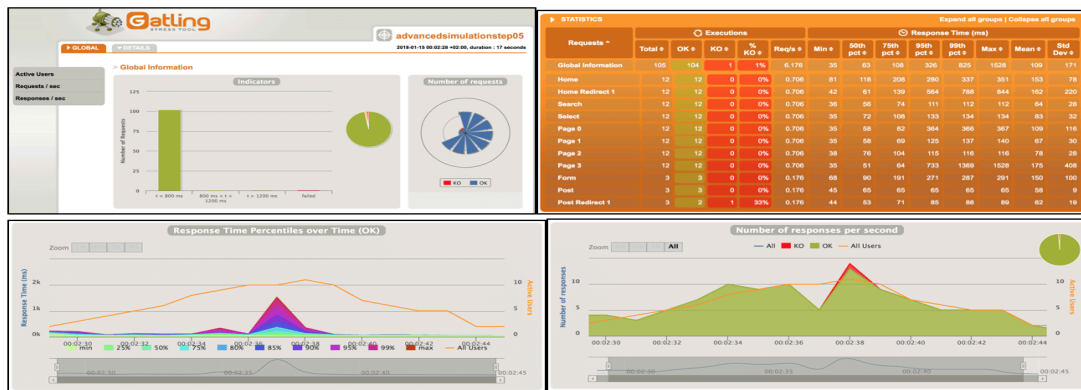
GATLING

An open-source load testing solution and performance testing framework that is used to simulate traffic and measure the performance of web applications, APIs, and other network services. It is available as an open-source as well as an enterprise variant called Gatling Frontline with more integrations. But all practical use cases can be worked upon using the Gatling community edition which is free and open-source. They create millions of virtual users. The Script is written in Scala (can be Java, kotlin).

Why Gatling

- High performance: Gatling is designed to handle high loads and concurrency, making it a good choice for testing the performance and scalability of web applications.
- Simulation-based approach: Gatling allows you to simulate user behavior and interactions with the application or website being tested, which can help identify performance bottlenecks and issues that might not be detected by other types of testing.
- Scripting in a familiar language: Gatling scripts are written in Scala, a popular programming language that is widely used in the industry. This makes it easy for developers and testers to write and maintain test scripts.

- Extensive reporting: Gatling provides detailed and customizable test reports, which can help identify performance issues and track improvements over time.



- Open-source and community-driven: Gatling is an open-source tool with an active and supportive community. This makes it easy to get help and find resources, and also allows for the development of custom plugins and extensions.

Scala

Scala smoothly integrates the features of object-oriented and functional languages. Scala Programming is based on Java. Scala is a pure object-oriented language in the sense that every value is an object. Scala is also a functional language in the sense that every function is a value and every value is an object so ultimately every function is an object. Scala is compiled into Java Byte Code which is executed by the Java Virtual Machine (JVM). This means that Scala and Java have a common runtime platform. You can easily move from Java to Scala.

Maven

Apache maven is a project building, and project management tool. Using Maven, we can able to build any java, scala-based projects. When a Maven project is created,

Maven creates a default project structure. A developer is only required to place files accordingly and he/she need not define any configuration in pom.xml.

Gatling Core Package

The **Gatling core package** is a fundamental part of the Gatling load testing framework. It provides the building blocks and infrastructure for developing and executing load-testing simulations in a scalable, efficient, and customizable way.

Some of the key components of the Gatling core package include:

- **Simulation:** Defines the simulation structure and scenarios, including the number of users, ramp-up time, and duration of the test.
- **Feeder:** Provides data sources for the simulation, such as CSV, JSON, or custom data sources.
- **Protocol:** Defines the protocol for the load test, such as HTTP or JDBC.
- **Assertion:** Defines the criteria for determining whether a load test is successful, such as response time or error rates.
- **Session:** Provides a mechanism for passing data between requests in a simulation.
- **Result:** Defines the structure of the test results, including metrics such as response time, throughput, and error rate.

Gatling HTTP Package

The **Gatling HTTP package** is a module of the Gatling load-testing framework that provides a set of tools and utilities for load-testing HTTP and HTTPS protocols.

Some of the key components of the Gatling HTTP package include:

- **HTTP protocol configuration:** Allows developers to configure HTTP-specific settings such as base URLs, headers, cookies, and authentication credentials.
- **HTTP request and response builders:** Provides a fluent API for building HTTP requests and processing HTTP responses.
- **HTTP checks:** Enables developers to define assertions for HTTP responses, such as status codes, headers, and response bodies.
- **Session handling:** Allows developers to extract values from HTTP responses and store them in sessions for use in subsequent requests.
- **Scenario and injection builders:** Provides a fluent API for defining HTTP scenarios and load injection profiles.

HTTP Headers

In the context of web development and communication between clients and servers, a header is a small piece of information that is sent as a part of an HTTP (Hypertext Transfer Protocol) request or response. The header contains additional information about the request or response that can be used by the client or server to interpret the message. HTTP headers are used for a variety of purposes, including:

- **Authentication:** HTTP headers can be used to provide authentication information, such as an access token, that allows the server to verify the identity of the client making the request.
- **Caching:** HTTP headers can be used to control the caching behavior of the client and server, allowing for more efficient data retrieval and reducing network traffic.
- **Compression:** HTTP headers can be used to specify whether the data being transferred should be compressed to reduce the amount of data sent over the network.
- **Content-type:** HTTP headers can be used to indicate the type of data being sent or received, such as text, HTML, JSON, or XML.
- **Cookies:** HTTP headers can be used to set and manage cookies, which are small pieces of data that are stored on the client side and used to track user behavior or session information.

THROUGHPUT

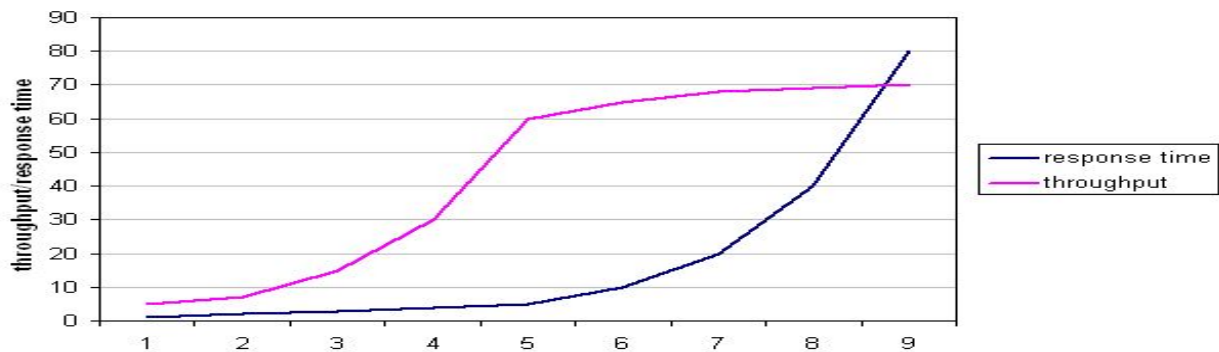
Throughput is a performance metric that measures the rate of transactions that can be processed by the system under test over a given period of time.

More specifically, throughput is defined as the number of requests completed per unit of time, typically measured in requests per second (RPS) or transactions per second (TPS).

For example, if a system can handle 100 requests per second, its throughput is 100 RPS. Throughput is an important performance metric because it helps to identify the maximum number of transactions that a system can handle before it becomes overloaded or starts to experience performance degradation.

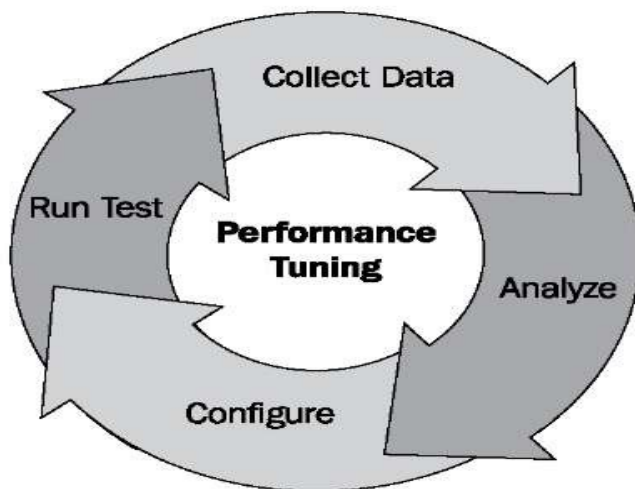
To calculate throughput in Gatling, the tool measures the number of requests that are completed within a given time interval and divides that by the duration of the interval.

For example, if 1,000 requests are completed in 10 seconds, the throughput would be calculated as 100 RPS ($1000 / 10 = 100$).



PERFORMANCE TUNING

Tuning is the procedure by which product performance is enhanced by setting different values to the parameters of the product, operating system, and other components. Performance tuning is the process of optimizing software applications or systems to ensure they run efficiently and effectively. The main goal of performance tuning is to improve the system's responsiveness, scalability, and stability while minimizing the resources it uses, such as CPU, memory, and disk I/O. Performance tuning involves identifying bottlenecks and areas of improvement in the system or application and making the necessary changes to optimize its performance.



BENCHMARKING

Benchmark testing is the process of load testing a component or an entire end to end IT system to determine the performance characteristics of the application. It is the process of measuring the performance of an application, system, or component against a standard or a reference point. The reference point could be a previous version of the same application, a competing product, or an industry standard. The primary goal of benchmarking is to identify the strengths and weaknesses of the system being tested and to compare them with the reference point. This helps in identifying areas that need improvement and in setting realistic goals for performance improvement.



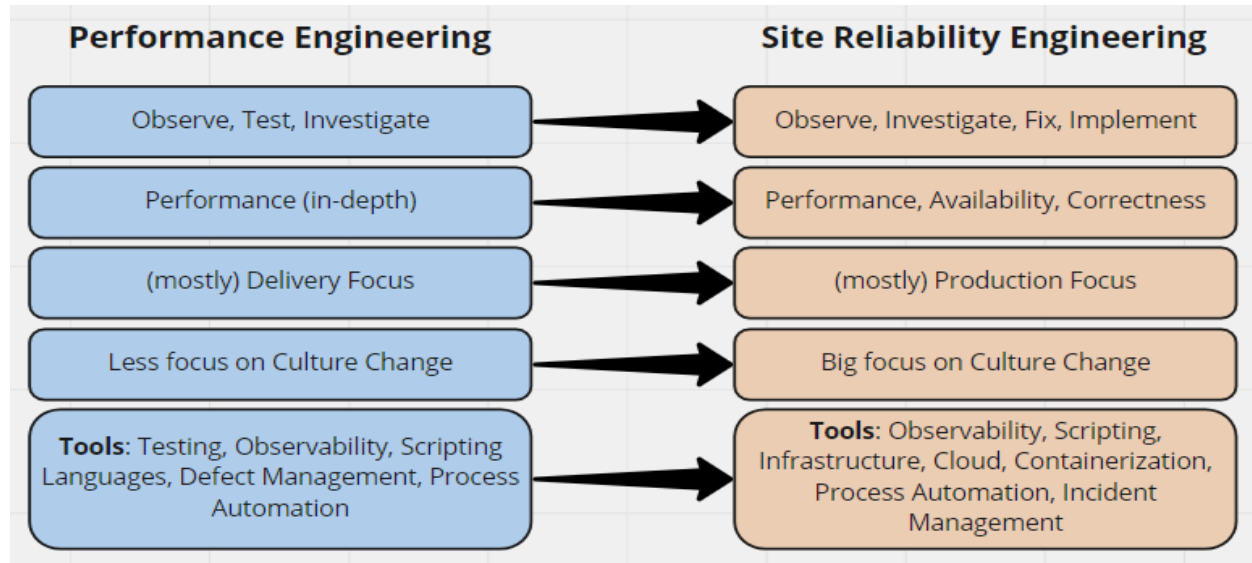
PERFORMANCE ENGINEERING

Performance engineering is a systematic approach to developing software applications to ensure they meet the expected performance objectives. It is a discipline that is focused on the architectural design, coding, and implementation choices that engineers make, including their technologies, practices, processes, and frameworks.

With a growing number of SRE teams continuously deploying applications, performance engineers must be tested regularly, as well as on-demand, to ensure the quality and stability of every additional integration.

The performance engineer suggests better architecture, code profiling, database optimization, analyzing business cases, monitoring, tuning, and performance forecasting. Performance engineering requires the active involvement of performance engineers in the SDLC for building a high-performance website or application. Performance engineering is taking performance concerns to the next level by helping developers to meet business case requirements and industry standards for speed, scalability, and sustainability.

PERFORMANCE ENGINEERING vs SRE



ASSERTION

In Gatling, an assertion is a way to check that certain conditions are met during the simulation. Assertions are typically used to verify the correctness of the application or website being tested, and to ensure that the simulated user behavior is consistent with the expected behavior.

Assertions can be defined for various aspects of the simulation, such as HTTP response codes, response times, response contents, or user sessions. An assertion can be added to a request using the `check` method, which takes an expression that evaluates to a `Validation` object.

1. Defining the scope of the assertion

- `global`: use statistics calculated from all requests.
- `forAll`: use statistics calculated for each individual request.
- `details(path)`: use statistics calculated from a group or a request. The path is defined like a Unix filesystem path.

2. Selecting the statistic

- `responseTime`: response time in milliseconds.

- `allRequests`: number of requests.
- `failedRequests`: number of failed requests.
- `successfulRequests`: number of successful requests.
- `requestsPerSec`: rate of requests per second.

3. Selecting the metric

Applicable to response time

- `min`: perform the assertion on the minimum of the metric.
- `max`: perform the assertion on the maximum of the metric.
- `mean`: perform the assertion on the mean of the metric.
- `stdDev`: perform the assertion on the standard deviation of the metric.
- `percentile1`: perform the assertion on the 1st percentile of the metric (default is 50th).
- `percentile2`: perform the assertion on the 2nd percentile of the metric (default is 75th).
- `percentile3`: perform the assertion on the 3rd percentile of the metric (default is 95th).
- `percentile4`: perform the assertion on the 4th percentile of the metric (default is 99th).
- `percentile(value: Double)`: perform the assertion on the given percentile of the metric. Parameter is a percentage, between 0 and 100.

Applicable to number of requests (all, failed or successful)

- `percent`: use the value as a percentage between 0 and 100.
- `count`: perform the assertion directly on the count of requests.

4. Defining the condition

- `lt(threshold)`: check that the value of the metric is less than the threshold.
- `lte(threshold)`: check that the value of the metric is less than or equal to the threshold.
- `gt(threshold)`: check that the value of the metric is greater than the threshold.
- `gte(threshold)`: check that the value of the metric is greater than or equal to the threshold.

- `between(thresholdMin, thresholdMax)`: check that the value of the metric is between two thresholds.
- `between(thresholdMin, thresholdMax, inclusive = false)`: same as above but doesn't include bounds
- `around(value, plusOrMinus)`: check that the value of the metric is around a target value plus or minus a given margin.
- `around(value, plusOrMinus, inclusive = false)`: same as above but doesn't include bounds.
- `deviatesAround(target, percentDeviationThreshold)`: check that metric is around a target value plus or minus a given relative margin
- `deviatesAround(target, percentDeviationThreshold, inclusive = false)`: same as above but doesn't include bounds
- `is(value)`: check that the value of the metric is equal to the given value.
- `in(sequence)`: check that the value of the metric is in a sequence.

```
setUp(population)
  .assertions(
    global.responseTime.max.lt(50),
    global.successfulRequests.percent.gt(95)
  )
```

SESSION

A session is a virtual user session that represents a user's interactions with the system being tested. It contains all the information about the user, including any data that the user might have submitted, as well as any responses received from the system.

Sessions are used to model user behavior in Gatling simulations. A typical Gatling simulation involves creating a scenario that specifies a series of actions to be performed by the virtual user, such as submitting a form or navigating to a new page. Each time the scenario is executed, a new session is created for the virtual user, which contains all the data needed to perform the specified actions.

Sessions are a fundamental concept in Gatling and are used extensively in Gatling simulations to model user behavior and interactions with the system being tested.

SCENARIO

These scenarios can be the result of measurements on the running application with analytic tools, or expected users' behavior of a new application. In any case, the creation of these scenarios is the key to meaningful results of the load test.

A scenario represents a typical user behavior. It's a workflow that virtual users will follow.

- Exec: The exec method is used to execute an action. Actions are usually requests that will be sent during the simulation. Any action that will be executed will be called with exec.
- Loops: Makes it browse several articles of each category,
- Conditions: To change its behavior depending on dynamic parameters,
- Pauses : To simulate a real user think-time.

INJECTIONS

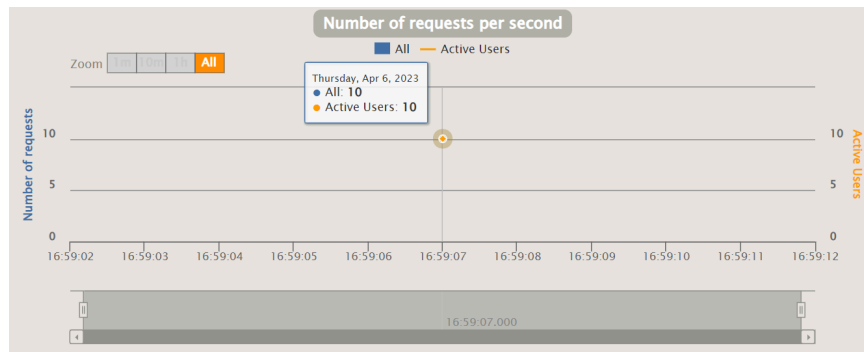
Injection is the process of generating user requests in a Gatling simulation. It determines how virtual users are added to a simulation over time, and can be configured to simulate different user loads.

Open Workload Models

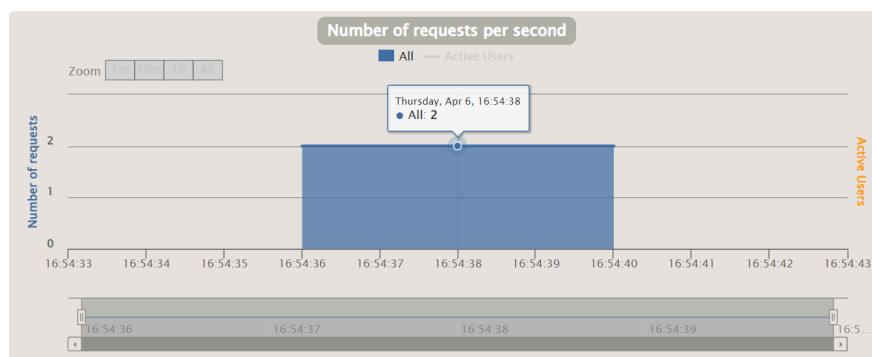
An open workload model is characterized by an unlimited number of Virtual users entering and leaving the system throughout the test execution. In this model, the number of VUs is not predefined, and new VUs are added as the test progresses. This model simulates real-world scenarios where the number of users accessing the system is unknown and can change at any given moment.

```
setUp(  
  scn.inject(  
    nothingFor(4) ,  
    atOnceUsers(10) ,  
    rampUsers(10).during(5) ,  
    constantUsersPerSec(20).during(15) ,  
    constantUsersPerSec(20).during(15).randomized ,  
    rampUsersPerSec(10).to(20).during(10.minutes) ,  
    rampUsersPerSec(10).to(20).during(10.minutes).randomized ,  
    heavisideUsers(5).during(5)  
  ).protocols(httpProtocol)  
)
```

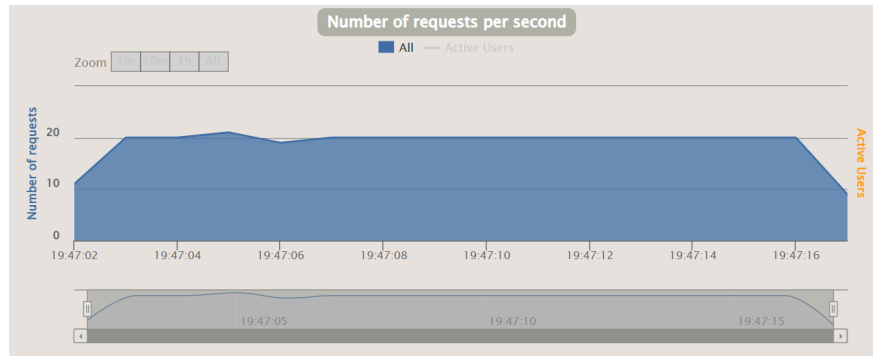
- `nothingFor(4)`: It is a pause method that adds a delay between injection steps. Gatling will wait for 4 seconds before starting to inject the next batch of users.
- `atOnceUsers(10)`: This method used to inject a specified number of virtual users into a simulation at once. This means that all the virtual users specified by `atOnceUsers` will start their sessions simultaneously. `atOnceUsers(10)` means that Gatling will inject 10 virtual users into the simulation at the start of the test.



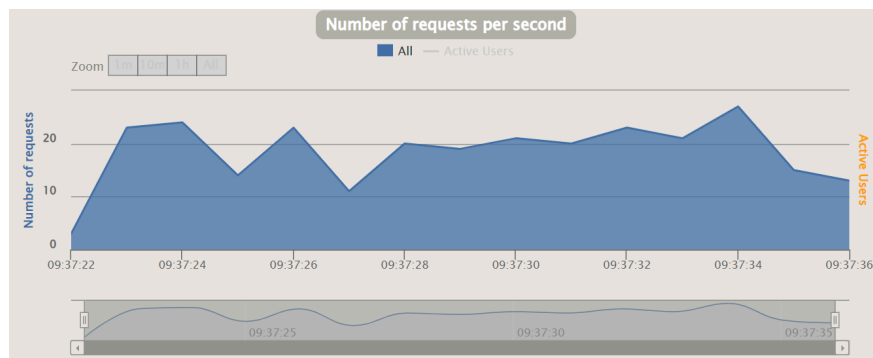
- `rampUsers(10).during(5)`: This method used to gradually increase the number of virtual users over a specified duration. The `during` method is used to specify the duration of the ramp-up period. `rampUsers(10).during(5)` means that Gatling will gradually increase the number of virtual users from 0 to 10 over a period of 5 seconds. So Total number of requests is 10 and Request per sec is 2 (users/sec).



- `constantUsersPerSec(20).during(15)`: This method used to maintain a constant rate of virtual users over a specified duration. The `during` method is used to specify the duration of the test. For example, `constantUsersPerSec(20).during(15)` means that Gatling will maintain a constant rate of 20 virtual users per second for a duration of 15 seconds. So the total number of requests is 300 (20 x 15) and Request per sec is 20 (users/sec).

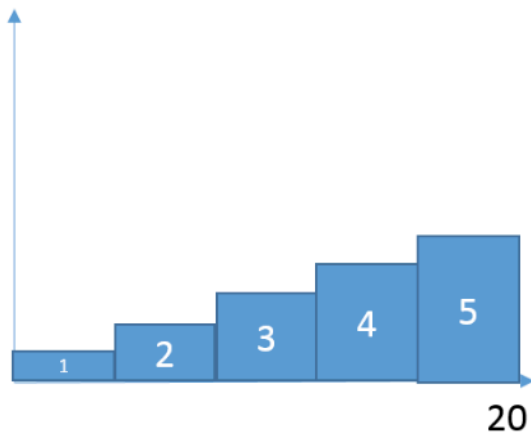


- `constantUsersPerSec(20).during(15).randomized`: A randomized method is used to introduce some randomness into the injection profile of virtual users. `constantUsersPerSec(20).during(15).randomized` means that Gatling will maintain a constant rate of 20 virtual users per second for a duration of 15 seconds, but will vary the number of users injected per second around that target rate.



- `rampUsersPerSec(1).to(5).during(20)`: This method in Gatling is used to gradually increase the number of users hitting the application over a period of time. During the ramp-up period, the number of users hitting the application will gradually increase until it reaches the target rate of 5 users per second. So the total number of requests is 60.

transactions



Each block is 4 transactions

- Block 1 = 1 tps repeated 4 times
- Block 2 = 2 tps repeated 4 times
- Block 3 = 3 tps repeated 4 times
- Block 4 = 4 tps repeated 4 times
- Block 5 = 5 tps repeated 4 times
- $5 \times 4 = 20$

User 1: 4×1

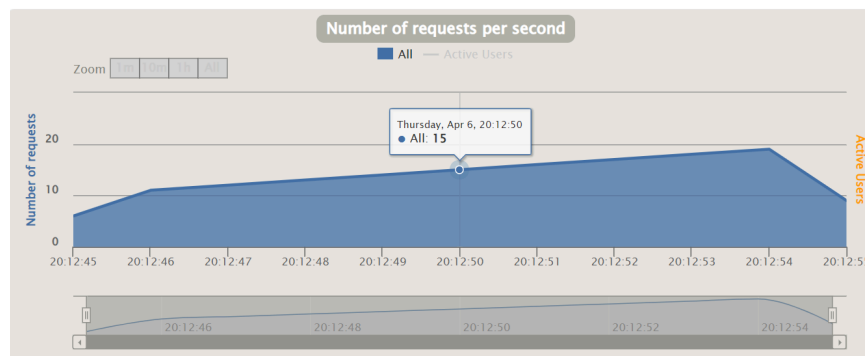
User 2: 4×2

User 3: 4×3

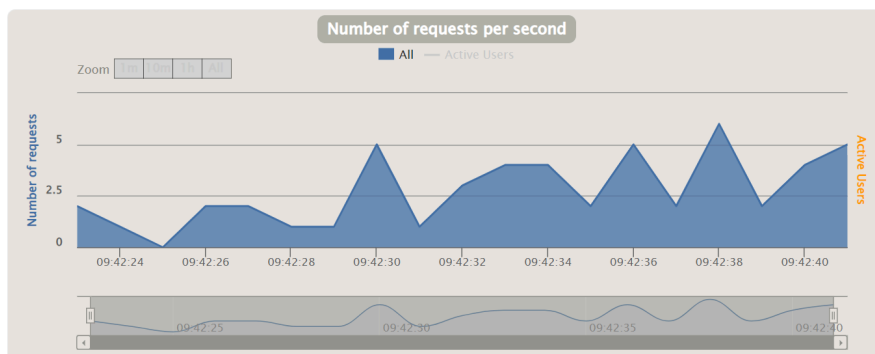
User 4: 4×4

User 5: 4×5

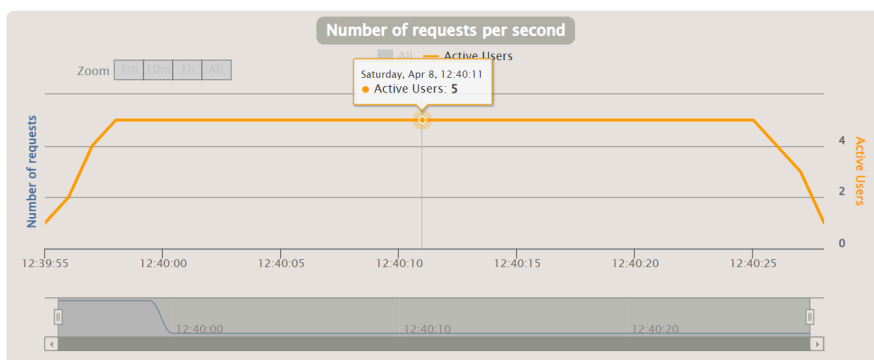
Total Request = User 1+User 2+User 3+User 4+User 5 = 60



- `rampUsersPerSec(10).to(20).during(10).randomized`: This injection profile will ramp up the number of users per second from 10 to 20 over a duration of 10 seconds, and the injection will be randomized.



- `heavisideUsers(5) during(5)`: The `heavisideUsers` injection profile in Gatling creates a step-shaped curve for the number of virtual users, where the number of users starts at 0 and rises gradually to a peak value, then stays at that peak value for some duration, and finally drops back down to 0. In `heavisideUsers(5) during(5)`, the injection profile would create a total of 5 virtual users that start immediately, then stay at 5 for 5 seconds, and then stop immediately. So the total number of requests sent by these 5 virtual users during this injection would depend on the specific scenario being executed.



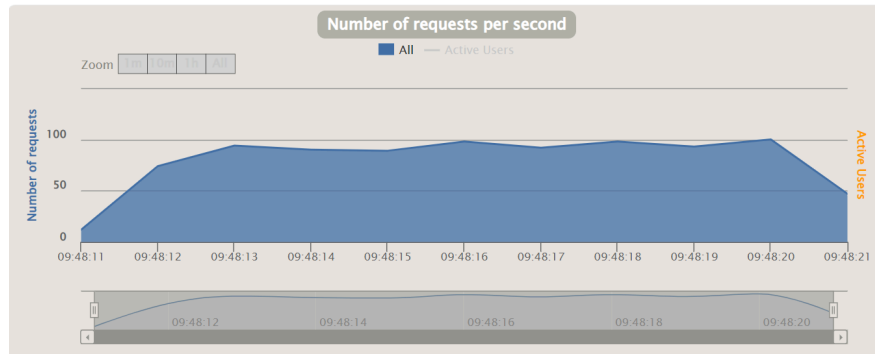
Closed Workload Models

A closed workload model is characterized by a fixed number of Virtual users that remain active throughout the test execution. In this model, the number of VUs is pre-defined, and once all VUs are active, no new VUs are added. This model is suitable for testing systems that have a predictable number of users accessing the system.

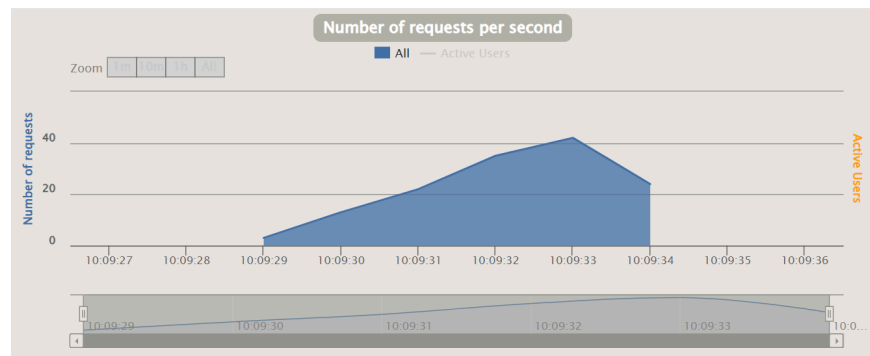
```
setUp(
  scn.inject(
    constantConcurrentUsers(10).during(10),
    rampConcurrentUsers(10).to(20).during(10)
  )
)
```

- `constantConcurrentUsers(10).during(10)`: This method is used to maintain a constant level of concurrent users throughout the simulation. In this

case, the simulation will have 10 concurrent users for a duration of 10 seconds. This means that at any given moment, there will be 10 virtual users executing requests concurrently. The total number of requests can be calculated by multiplying the number of concurrent users with the number of requests sent per second.



- `rampConcurrentUsers(1).to(5).during(5):` This means that the number of concurrent users will gradually increase from 1 to 5 over a duration of 5 seconds. The total number of requests that will be generated during this period will depend on the scenario definition, i.e., the number of requests made by each virtual user per second, the think times, and other settings.



PERCENTILE

Percentile is a statistical measure that indicates the value below which a given percentage of observations in a group of observations falls. It is used to measure the performance of a system by measuring the response times at different points during a load test.

For example, the 95th percentile of response times means that 95% of the requests were responded to within that time or less. The 99th percentile means that 99% of requests were responded to within that time or less.

Let's say, we run a PT script that will hit a single API 10 times and we have the below response time for each request:

250, 340, 200, 193, 377, 450, 280, 310, 500, 245

So, we arrange them in ascending order:

193, 200, 245, 250, 280, 310, 340, 377, 450, 500

Here, the 95th percentile will be **450 ms** (9 requests have less than or equal to 450 ms, remaining one request have greater than 450 ms).

In Gatling, Percentile is a built-in metric that can be used to measure the performance of a system during a load test. It can be used to calculate the response time of different percentiles and to analyze the performance of the system under different load conditions. Gatling provides a way to configure the percentiles in the simulation configuration file (`gatling.conf`).

STATISTICS														Expand all groups Collapse all groups	
Requests ^	Executions					Response Time (ms)									
	Total ↕	OK ↕	KO ↕	% KO ↕	Cnt/s ↕	Min ↕	50th pct ↕	75th pct ↕	95th pct ↕	99th pct ↕	Max ↕	Mean ↕	Std Dev ↕		
Global Information	139	139	0	0%	23.167	85	105	110	122	140	185	105	12		
get user request	139	139	0	0%	23.167	85	105	110	122	140	185	105	12		

INFLUXDB & GRAFANA WITH GATLING

InfluxDB and Grafana are popular tools for storing and visualizing time-series data, and they can be integrated with Gatling to store and display performance test results. Here are the high-level steps to integrate InfluxDB and Grafana with Gatling:

1. Install InfluxDB and Grafana: Download and install InfluxDB and Grafana on your system.
2. Configure InfluxDB: Configure InfluxDB to store the performance metrics that Gatling generates during the test execution.

3. Add InfluxDB dependencies: Add the necessary dependencies to the Gatling project to enable the integration with InfluxDB.
4. Configure Gatling: Configure Gatling to send the performance metrics to InfluxDB during the test execution.
5. Start the test: Start the Gatling test execution.
6. Visualize results in Grafana: Use Grafana to create dashboards and visualize the performance test results.

