**Test Plan**

**Overview:** The goal is to test certain high volume github user requests and gauge at what point the response times degrade based on increased requests. Also to gauge the scalability of the system with increasing user load as well as stability over an extended period of time.

**Objectives:** The performance testing objectives are to discover at which point the various test scenarios begin to show degradation in response times of the specified user scenarios.

**Tools selection:**

1) Gatling was chosen as a stable open source tool that has reliable metrics gathering as well as graphing. Gatling has decent status monitoring and reporting.

2) Ruby was used to generate testing assets such as data input files. Github\_api gem was used to create github api calls.

3) Charles Web Debugging Proxy tool used to analyze http traffic.

4) Splunk is the tool of choice for log and system analysis which give time sorted results across search indexes.

**Testing environment setup.**

Create an EC2 instance.

Install Gatling

Install Splunk

**Generate testing assets:**

Generate acceptable dynamic user input data sets. Ruby scripts were used to build testing assets. Use large and random data sets to guard against caching.

Users.csv: A list of acceptable users to login to the system

Branches.csv: A list of acceptable branches to perform test scenario operations on.

Edit\_files.csv: A list of acceptable files to create on the target repositories.

**Performance Test Execution:**

Identify performance test scenarios.

Create baselines. Identify typical response times for each scenario without load.

Build test scenarios using Gatling.

Build test assets to enable parameterizing of functions and dynamic inputs.

Determine reasonable user ramping and execute load scenarios.

Monitor running tests and logs for failures.

Analyze results.

**Deliverables:**

Performance Test Plan (this document)

Performance Test Results (found in results

Performance Test Metrics

Performance Analysis, comments, and recommendations

**Test Scenarios:**

Test Scenario 1:

Login with x number of users and commit changes to the master branch on the docker repo.

Test Scenario 2:

Login with user, create a branch off of master, commit a change (new file or edit file) and create a pull request for the commit to master.

Test Scenario 3:

Create x number of comments on an existing pull request with (x) number of users.

Test Scenario 4:

(x) number of users download (x) number of repos.

**Run baselines:**

Capture baseline results for test scenarios without load to record response times which will let us interpret performance degradation under load.

**Performance Test Types:**

All four test scenarios will be exercised for all performance test types.

Baseline Test: Capture baseline results for test scenarios without load to record response times which will let us interpret performance degradation under load. This test will ramp virtual users from 1-5 over a period of 2 minutes to get an average response time.

Regression Test: This will be the same scenario as the baseline test and will be run on every subsequent build to ensure that there has not been any performance degradation with respect to response times between build. This will allow us to track performance creep between builds/releases. Ramp virtual users from 1-5 over a period of 2 minutes with any outside load absent.

Stress Test: This test will evaluate at what point the response time starts to diminish and eventually becomes unacceptable to the given requirements. The test will ramp from 1 – 30 users with a maximum of 10 concurrent users. The test will continue until 100 virtual users have been injected into the system. The parameters will be adjusted as necessary to induce the desired results of the test.

Stabliity (Spike) Test: This will be a prolonged test that will increase and decrease concurrent user load. The expectation is that response times will increase and decrease with increase and decrease in user requests, respectively. The analysis here will be to see if the response time return to normal as the load returns to normal and that underlying services are resilient to the extended periods of load. We will also be recording how many errors (and error type) occur at peak loads and if the errors continue as load reduces. This test will run for 12 hours, cycling between 1 and 30 users throughout the period of the test.

Scalability Test: This test will also be a prolonged test with the goal to see what the maximum number of user requests can be handled before reaching failure (defined in requirements). This test will ramp slowly from 1 to 100 concurrent virtual users to identify at what number of concurrent virtual users the system can handle.

**Performance Goals:**

These are the things that we should have a requirement or expectation of and determine each with testing and analysis.

What is current response time?

What is an acceptable response time?

What percent of user requests must meet the acceptable response time?

What is the baseline response time multiplier that the system must be capable of handling?

The spike ratio that the over all stability of the system must be capable of handling?

What ratio and magnitude of bursts should the system be able to handle (adjust stability test parameters)?

**Test Metrics to collect**

Response times of requests. Gatling handles this built in.

Concurrent users at any give time. Gatling built in.

Backend system stats (memory, cpu, disk I/O, vmstat, iostat).

Network performance statistics (netstat).

Backend database read/write statistics.

Kernel messages

Tcpdump output

\*Note backend statistics are available for this exercise but are given for completeness

For Scalability, all relevant application and system logs should be monitored. A good log collection tool which interleaves with respect to time (i.e Splunk) is extremely useful here. Similarly, we can configure all systems to forward their metrics to Splunk as well as have databases forward their read/write activity to Splunk.

**Analysis & Reporting:**

Do not assume a bottleneck root cause unless proven. Do identify bottleneck symptoms.

Performance engineers will execute all created scripts. The scripts will be executed against the system at least 3x to help rule out test system anomalies. The scripts will be run after any hardware, software, or any general infrastructure changes are inserted into the system.

Analysis of results and actual test results will be maintained over time to allow for deeper analysis over time. All stakeholders will be notified of each new addition to the test results repository.