Load Testing

"*Load testing is performed to determine a system's behavior under both normal and anticipated peak load conditions*." - Load testing - Wikipedia

A load test is designed to determine how a system behaves under expected normal and peak workloads. Specifically its main purpose is to confirm if a system can handle the expected load level. Depending on the target system this could be concurrent users, requests per second or data size.

Why Load Testing

The main objective is to prove the system can behave normally under the expected normal load before releasing it to production. The criteria that define "behave normally" will depend on your target, this may be as simple as "the system remains available", but it could also include meeting a response time SLA or error rate.

Additionally, the results of a load test can also be used as data to help with capacity planning and calculating scalability.

Load Testing Design Blocks

There are a number of basic components that are required to carry out a load test.

1. In order to have meaningful results the system needs to be tested in a production-like environment with a network and hardware which closely resembles the expected deployment environment.
2. The load test will consist of a module which simulates user activity. Of course the composition of this "user activity" will vary based on the type of application being tested. For example, an e-commerce website might simulate user browsing and purchasing items, but an IoT data ingestion pipeline would simulate a stream of device readings. Please ensure the simulation is as close to real activity as possible, and consider not just volume but also patterns and variability. For example, if the simulator data is too uniform or predictable, then cache/hit ratios may impact your results.

3. The load test will be initiated from a component external to the target system which can control the amount of load applied. This can be a single agent, but may need to scaled to multiple agents in order to achieve higher levels of activity.
4. Although not required to run a load test, it is advisable to have monitoring and/or logging in place to be able to measure the impact of the test and discover potential bottlenecks.

Applying the Load Testing

Planning

1. **Identify key scenarios to measure -** Gather these scenarios from Product Owner, they should provide a representative sample of real world traffic. The key activity of this phase is to agree on and define the load test cases.
2. **Determine expected normal and peak load for the scenarios -** Determine a load level such as concurrent users or requests per second to find the size of the load test you will run.
3. **Identify success criteria metrics -** These may be on testing side such as response time and error rate, or they may be on the system side such as CPU and memory usage.
4. **Agree on test matrix -** Which load test cases should be run for which combinations of input parameters.
5. **Select the right tool -** Many frameworks exist for load testing so consider if features and limitations are suitable for your needs (Some popular tools are listed below). This may also include development of a custom load test client, see Preparation phase below.
6. **Observability -** Determine which metrics need to gathered to gain insight into throughput, latency, resource utilization, etc.
7. **Scalability -** Determine the amount of scale needed by load generator, workload application, CPU, Memory, and network components needed to achieve testing goals. The use of kubernetes on the cloud can be used to make testing infinitely scalable.

Preparation

The key activity is to replace the end user client with a test bench that simulates one or more instances of the original client. For standard 3rd party tools it may suffice to configure the existing test UI before initiating the load tests.

If a custom client is used, code development will be required:

1. **Custom development -** Design for minimal impact/overhead. Be sure to capture only those features of the production client that are relevant from a load perspective. Does it matter if the same test is duplicated, or must the workload be unique for each test? Can all tests be run under the same user context?
2. **Test environment -** Create test environment that resembles production environment. This includes the platform as well as external systems, e.g., data sources.
3. **Security contexts -** Be sure to have all requisite security contexts for the test environment. Automation like pipelines may require special setup, e.g., OAuth2 client credential flow instead of auth code flow, because interactive login is replaced by non-interactive. Allow planning leeway in case admin approval is required for new security contexts.
4. **Test data strategy -** Make sure that output data format (ascii/binary/...) is compatible with whatever analysis tool is used in the analysis phase. This also includes storage areas (local/cloud/...), which may trigger new security contexts. Bear in mind that it may be necessary to collect data from sources external to the application to correlate potential performance issues with the application behavior. This includes platform and network metrics. Make sure to collect data that covers analysis needs (statistical measures, distributions, graphs, etc.).
5. **Automation -** Repeatability is critical. It must be possible to re-run a given test multiple times to verify consistency and resilience of the application itself and the underlying platform. Pipelines are recommended whenever possible. Evaluate whether load tests should be run as part of the PR strategy.
6. **Test client debugging -** All test modules should be carefully debugged to ensure that the execution phase progresses smoothly.
7. **Test client validation -** All test modules should be validated for extreme values of the input parameters. This reduces the risk of running into unexpected difficulties when stepping through the full test matrix during the execution phase.

Execution

It is recommended to use an existing testing framework (see below). These tools will provide a method of both specifying the user activity scenarios and how to execute those at load. Depending on the situation, it may be advisable to coordinate testing activities with the platform operations team.

It is common to slowly ramp up to your desired load to better replicate real world behavior. Once you have reached your defined workload, maintain this level long enough to see if your system stabilizes. To finish up the test you should also ramp to see record how the system slows down as well.

You should also consider the origin of your load test traffic. Depending on the scope of the target system you may want to initiate from a different location to better replicate real world traffic such as from a different region.

> **Note:** Before starting please be aware of any restrictions on your network such as DDOS protection where you may need to notify a network administrator or apply for an exemption.
>
> **Note:** In general, the preferred approach to load testing would be the usage of a standard test framework such as the ones discussed below. There are cases, however, where a custom test client may be advantageous. Examples include batch oriented workloads that can be run under a single security context and the same test data can be re-used for multiple load tests. In such a scenario it may be beneficial to develop a custom script that can be used interactively as well as non-interactively.

Analysis

The analysis phase represents the work that brings all previous activities together:

- Set aside time to allow for collection of new test data based on the analysis of the load tests.
- Correlate application metrics and platform metrics to identify potential pitfalls and bottlenecks.
- Include business stakeholders early in the analysis phase to validate application findings. Include platform operations to validate platform findings.

Report Writing

Summarize your findings from the analysis phase. Be sure to include application and platform enhancement suggestions, if any.

Further Testing

After completing your load test you should be set up to continue on to additional related testing such as;

- **Soak Testing -** Also known as **Endurance Testing**. Performing a load test over an extended period of time to ensure long term stability.
- **Stress Testing -** Gradually increasing the load to find the limits of the system and identify the maximum capacity.
- **Spike Testing -** Introduce a sharp short-term increase into the load scenarios.

- **Scalability Testing** - Re-testing of a system as your expand horizontally or vertically to measure how it scales.
- **Distributed Testing** - Distributed testing allows you to leverage the power of multiple machines to perform larger or more in-depth tests faster. Is necessary when a fully optimized node cannot produce the load required by your extremely large test.

Load Generation Testing Frameworks and Tools

Here are a few popular load testing frameworks you may consider, and the languages used to define your scenarios.

- **Azure Load Testing** (https://learn.microsoft.com/en-us/azure/load-testing/) - Managed platform for running load tests on Azure. It allows to run and monitor tests automatically, source secrets from the KeyVault, generate traffic at scale, and load test Azure private endpoints. In the simple case, it executes load tests with HTTP GET traffic to a given endpoint. For the more complex cases, you can upload your own **JMeter scenarios**.
- **JMeter** (https://github.com/apache/jmeter) - Has built in patterns to test without coding, but can be extended with Java.
- **Artillery** (https://artillery.io/) - Write your scenarios in Javascript, executes a node application.
- **Gatling** (https://gatling.io/) - Write your scenarios in Scala with their DSL.
- **Locust** (https://locust.io/) - Write your scenarios in Python using the concept of concurrent user activity.
- **K6** (https://k6.io/) - Write your test scenarios in Javascript, available as open source kubernetes operator, open source Docker image, or as SaaS. Particularly useful for distributed load testing. Integrates easily with prometheus.
- **NBomber** (https://nbomber.com/) - Write your test scenarios in C# or F#, available integration with test runners (NUnit/xUnit).
- **WebValidate** (https://github.com/microsoft/webvalidate) - Web request validation tool used to run end-to-end tests and long-running performance and availability tests.

Sample Workload Applications

In the case where a specific workload application is not being provided and the focus is instead on the system, here are a few popular sample workload applications you may consider.

- **HttpBin** (Python, GoLang) - Supports variety of endpoint types and language implementations. Can echo data used in request.

- **NGSA** (Java, C#) - Intended for Kubernetes Platform and Monitoring Testing. Built on top of IMDB data store with many CRUD endpoints available. Does not need to have a live database connection.
- **MockBin** (https://github.com/Kong/mockbin) - Allows you to generate custom endpoints to test, mock, and track HTTP requests & responses between libraries, sockets and APIs.

Conclusion

A load test is critical step to understand if a target system will be reliable under the expected real world traffic.

Of course, it's only as good as your ability to predict the expected load, so it's important to follow up with other further testing to truly understand how your system behaves in different situations.

Resources

List additional readings about this test type for those that would like to dive deeper.

- Microsoft Azure Well-Architected Framework > Load Testing

Last update: August 22, 2024