

Identifying Service Performance Metrics

Overview

Scalability and performance testing provides meaningful business value when it is performed using a consistent, repeatable, and measurable methodology.

A performance test should do more than determine whether a service is "fast enough." It should provide actionable information about how the service behaves under different workloads, how much traffic it can support, and where performance or scalability begins to degrade.

A complete performance testing methodology should address three fundamental areas:

1. **Selecting representative test cases**
2. **Executing accurate and repeatable test runs**
3. **Analyzing the results to identify scalability, capacity, and performance characteristics**

These areas work together to provide a clear understanding of how a service behaves under realistic and increasing workloads.

1. Select the Right Test Cases

The first step in performance testing is determining what should be tested.

Not all services behave the same way, and therefore not all services should be tested using the same workload.

For example, a service that handles a high volume of small requests may require a different test strategy from a service that receives large messages periodically.

Consider the following examples:

High-volume service

Small requests
High request frequency
Many concurrent users

The primary concern may be:

- Maximum throughput.
- Transactions per second (TPS).
- Concurrent request capacity.
- Response time under sustained load.

Large-message service

Large request payloads
Lower request frequency
Higher processing cost per request

The primary concerns may instead include:

- Response time.
- Network throughput.
- Processing time.
- Memory consumption.
- Maximum supported message size.

Multiple-interface service

A service may expose several interfaces or API operations, each with different performance characteristics.

For example:

POST /login
GET /products

POST /cart
POST /checkout

Testing only one endpoint may not accurately represent the workload generated by a real user.

The selected test cases should therefore represent the **actual business and user workflows** that the service is expected to support.

2. Define Representative Workloads

After identifying the relevant test cases, define the workload that will be applied to each test.

Important workload dimensions include:

- Number of concurrent users or requests.
- Request frequency.
- Message or payload size.
- Message complexity.
- Transaction mix.
- Duration of the test.
- Ramp-up and ramp-down behavior.
- Environment and configuration.

For example, a test plan could include:

Scenario	Concurrent Users	Payload	Duration
Baseline	10	Small	10 min
Normal Load	100	Small	30 min
High Load	500	Small	30 min
Large Payload	100	Large	30 min
Stress	1,000	Small	Until degradation

The objective is not simply to generate as much traffic as possible.

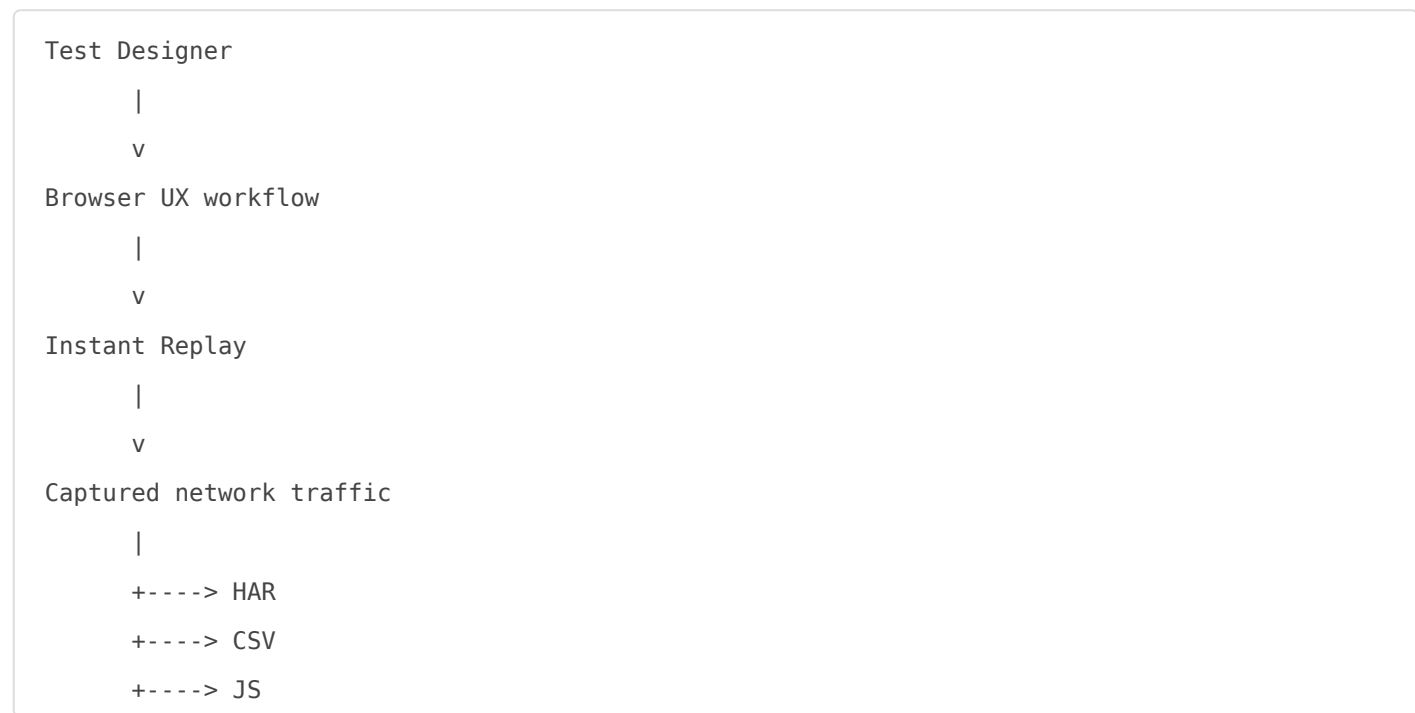
The objective is to understand **how the service behaves as the workload changes**.

3. Capture Real Application Traffic

Whenever possible, performance scenarios should be based on realistic application behavior.

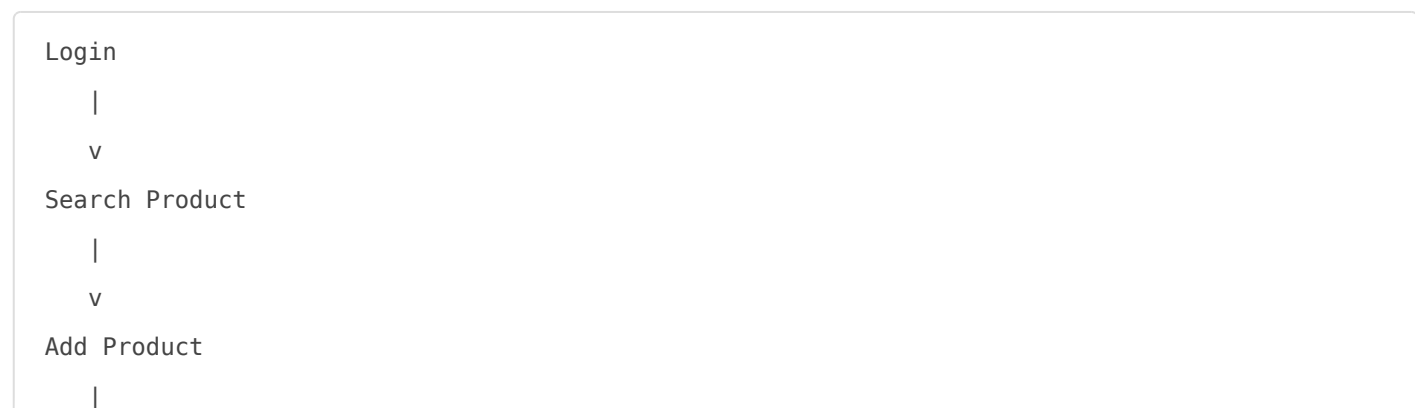
For browser-based workflows, **Test Designer Instant Replay** can be used to capture the network activity generated during recording or playback.

A typical workflow is:



The captured traffic can help identify the API requests associated with a real user workflow.

For example:



The corresponding API traffic can then be reviewed and, when appropriate, reused to build protocol-level performance scenarios.

This approach helps maintain a connection between **real user behavior** and the workload used during performance testing.

4. Execute Accurate and Repeatable Test Runs

A single performance test run is rarely sufficient to understand scalability.

Performance results can be affected by many factors, including:

- Current system load.
- Network conditions.
- Test environment configuration.
- Database activity.
- Application state.
- Test data.
- Infrastructure resources.
- Number of concurrent requests.

For this reason, performance testing should consist of multiple controlled and repeatable executions.

A typical progression may look like:

```
Run 1   → Baseline
Run 2   → Normal load
Run 3   → Increased load
Run 4   → High load
Run 5   → Stress
Run 6   → Repeat high load
```

Repeated executions make it easier to distinguish consistent system behavior from temporary variations.

Manual or ad-hoc recording of results is not sufficient for large-scale performance testing. Test execution and data collection should be automated whenever possible.

Automation provides:

- Consistent execution.
 - Repeatable workloads.
 - Standardized measurements.
 - Automated result collection.
 - Easier comparison between runs.
 - Reduced human error.
-

5. Define the Performance Metrics

A performance test is only useful if the results are measured using clearly defined metrics.

The following metrics are commonly used to evaluate service performance.

Transactions Per Second (TPS)

TPS measures the number of transactions processed by the service per second.

For example:

TPS = 1,500

means that the test generated approximately 1,500 transactions per second.

TPS is particularly useful when evaluating service throughput and capacity.

Response Time

Response time measures how long it takes for a request to complete.

For example:

Average Response Time = 250 ms

Average response time is useful, but it should not be the only response-time metric considered.

Averages can hide slow requests.

Percentile Response Times

Percentiles provide a better understanding of the distribution of response times.

Common metrics include:

- **p50** — 50% of requests completed within this time.
- **p90** — 90% of requests completed within this time.
- **p95** — 95% of requests completed within this time.
- **p99** — 99% of requests completed within this time.

For example:

p50 = 200 ms
p90 = 350 ms
p95 = 500 ms
p99 = 1,200 ms

This tells us that while most requests are relatively fast, a small percentage are taking significantly longer.

Error Rate

Error rate measures the percentage of requests that fail.

For example:

Total Requests = 1,000,000
Failed Requests = 2,000

Error Rate = 0.2%

Errors should be analyzed alongside throughput and response time.

A system that processes 5,000 TPS but returns a high percentage of errors cannot be considered healthy.

Concurrent Users or Requests

Concurrency represents how many users or requests are active at the same time.

Increasing concurrency is one of the primary ways to evaluate scalability.

For example:

100 users	→ 1,000 TPS
500 users	→ 3,500 TPS
1,000 users	→ 5,000 TPS
2,000 users	→ 5,100 TPS

In this example, increasing concurrency beyond 1,000 users produces very little additional throughput.

That may indicate that the system is approaching a capacity limit.

Resource Utilization

Performance testing should also consider the resources supporting the service.

Depending on the architecture, relevant measurements may include:

- CPU utilization.
- Memory utilization.
- Disk utilization.
- Network throughput.
- Database utilization.
- Connection pools.
- Thread pools.
- Queue depth.

These metrics help explain **why** performance changes as load increases.

6. Understand Scalability

Scalability describes how the system responds when the workload increases.

Consider this example:

Concurrent Users	TPS
10	500
50	1,200
100	2,000
500	3,500
1,000	4,000
2,000	4,050

Initially, increasing concurrency produces a significant increase in throughput.

However, after approximately 1,000 users, TPS begins to level off.

This indicates that the service may be approaching its practical capacity.

At the same time, response times may begin to increase:

Users	TPS	p95 Response Time
100	2,000	300 ms
500	3,500	450 ms
1,000	4,000	800 ms
2,000	4,050	2,500 ms

Although TPS continues to increase slightly, response time has degraded significantly.

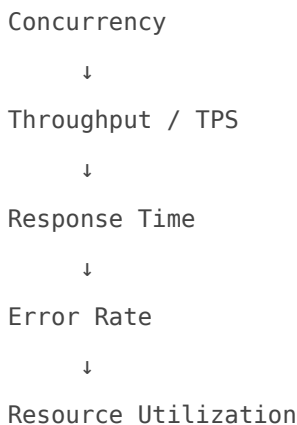
This is an important scalability finding.

7. Analyze the Relationship Between Workload and Performance

The objective of performance testing is not simply to collect numbers.

The objective is to understand the relationship between workload and system behavior.

Important relationships include:



For example, increasing concurrency may initially improve throughput.

Eventually, the system may reach a saturation point where:

- TPS stops increasing.
- Response times increase.
- Error rates increase.
- CPU or memory utilization becomes high.
- Database or network resources become constrained.

This point is often more important than the maximum TPS observed during the test.

8. Identify the Service Capacity

Performance testing should help establish the practical capacity of the service.

For example:

Maximum observed throughput: 4,100 TPS

But at 4,100 TPS:

p95 response time = 3 seconds

Error rate = 4%

It would be misleading to state that the service supports 4,100 TPS as a healthy operating point.

Instead, the team might determine that:

Recommended operating capacity:

3,500 TPS

p95 response time:

< 500 ms

Error rate:

< 1%

This provides a much more useful capacity target.

9. Compare Different Environments and

Configurations

Performance testing should also be performed across relevant environments and configurations.

For example:

```
QA
|
+---- Configuration A
|
+---- Configuration B

UAT
|
+---- Configuration A
|
+---- Configuration B

Production-like
|
+---- Configuration A
|
+---- Configuration B
```

The same workload can then be used to determine whether differences in configuration or infrastructure affect:

- TPS.
- Response time.
- Error rate.
- Resource utilization.
- Maximum capacity.

For comparisons to be meaningful, the test conditions should remain as consistent as possible.

10. Establish a Baseline

Before evaluating whether a change improves or degrades performance, establish a baseline.

A baseline should capture the expected behavior of the service under a known workload.

For example:

Baseline

Concurrency: 500

TPS: 3,500

p95: 450 ms

Error Rate: 0.2%

CPU: 65%

After an application or infrastructure change, the same workload can be executed again.

New Version

Concurrency: 500

TPS: 3,200

p95: 700 ms

Error Rate: 0.8%

CPU: 82%

The comparison immediately shows that the new version performs worse under the same workload.

This type of baseline comparison is much more useful than simply saying that a test "passed."

11. Turn Results into Actionable Knowledge

The final step is to interpret the results and produce conclusions that can support technical or business decisions.

A useful performance conclusion should answer questions such as:

- What workload can the service currently support?

- What is the maximum sustainable throughput?
- At what point does response time degrade?
- When does the error rate become unacceptable?
- Which resource becomes the bottleneck?
- Does the service scale as concurrency increases?
- How does payload size affect performance?
- How does message complexity affect performance?
- Does the service behave consistently across environments?
- Did a new application version improve or degrade performance?

For example:

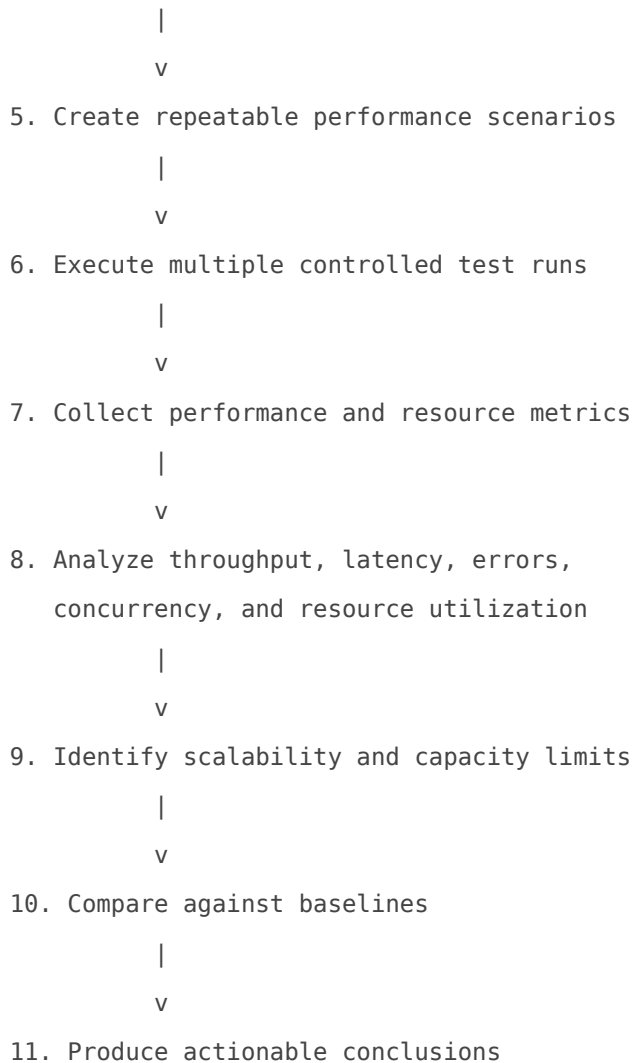
“ The service scales from 100 to approximately 500 concurrent users, increasing throughput from 1,800 to 3,500 TPS. Beyond 500 users, throughput increases only marginally while p95 response time increases significantly. CPU utilization reaches 90% at approximately 1,000 concurrent users, indicating CPU saturation as the primary scalability constraint.

This is actionable information because it provides a basis for deciding whether to optimize the application, increase infrastructure capacity, or establish a lower operating limit.

Performance Testing Methodology

The complete methodology can be summarized as follows:

1. Identify user workflows
|
v
2. Select representative test cases
|
v
3. Define workload characteristics
|
v
4. Capture realistic API/network traffic



Appvance Performance Testing Approach

The Appvance approach combines realistic user workflows with automated performance testing and measurable service-level results.

A typical workflow can begin with **Test Designer**, where the user journey is recorded or executed.

Instant Replay can then capture the network traffic generated by that workflow.

The captured traffic can be reviewed or exported and used as a foundation for protocol-level testing.

This provides a connection between:

User Experience

|

v

Test Designer

|

v

Network/API Traffic

|

v

Instant Replay

|

v

Protocol-Level Test

|

v

Performance Test

|

v

Metrics and Analysis

|

v

Scalability / Capacity Findings

This approach allows teams to evaluate both the **user-facing workflow** and the **service behavior behind that workflow**.

Key Takeaways

Effective scalability and performance testing requires more than generating a large number of requests.

A successful methodology should:

1. **Test realistic user and service workflows.**
2. **Use representative workloads and data.**
3. **Automate and repeat test execution.**
4. **Measure consistent and meaningful metrics.**
5. **Evaluate throughput as workload increases.**
6. **Monitor response-time percentiles and error rates.**

7. **Monitor infrastructure and service resources.**
8. **Identify capacity and saturation points.**
9. **Compare results against established baselines.**
10. **Turn test results into actionable technical and business conclusions.**

The ultimate goal is not simply to determine whether a service can process a particular number of requests.

The goal is to understand **how the service behaves as demand increases, where its limits are, and what actions are required to maintain acceptable performance and scalability.**

Last updated: August 12, 2026

For feedback or questions about this documentation, contact the appropriate Appvance documentation or support team.

Revision #5

Created 2026-07-31 15:09:09 UTC by Admin

Updated 2026-10-15 15:54:41 UTC by mmora