2.6 - Scalability and Performance Test Methodology

Scalability and Performance Test Methodology — Appvance IQ documentation.

- Scalability and Performance Test Methodology
- Applying the Method
- Background and Goals
- Add Calibration Testing to the Performance Test Plan
- Calibration What-if Chart
- Identifying Service Performance Metrics
- Identify Test Environment Hardware and Software
- Method for Black Box and White Box Tests
- Scalability Index
- Scalability and Performance Kit Contents
- Appvance Method
- Understanding the Tests
- Understand TPS in AIQ Performance Testing
- User Goal Oriented Testing (UGOT)
- Performance Testing Primer for AIQ
- Test Designer Performance Scalability: Maximizing the number of concurrent browsers per node

Scalability and Performance Test Methodology

Use this article to plan AIQ performance tests that define goals, load model, environment readiness, execution controls, and result interpretation before a test is run.

What changed in this article

- Kept the title but refreshed the methodology around current AIQ performance testing practices.
- Added checks for business objective, workload model, test data, environment parity, monitoring, ramp-up, duration, success criteria, and result review.
- Added guidance to separate application bottlenecks from AIQ/Test Node capacity constraints.

Validation workflow

1. Define the performance question, workload model, success criteria, and monitoring plan before building the test.
2. Validate data, environment readiness, Test Node capacity, and baseline behavior with a small run.
3. Run the planned test and review throughput, response time, errors, saturation, and evidence before making conclusions.

Review note: This page now reads as a practical methodology checklist instead of a generic performance article.

Last updated: October 2, 2026

If you have feedback on this topic, contact documentation@appvance.ai.

Applying the Method

The Appvance method evaluates scalability, performance, and developer productivity across different implementation approaches for Internet application services. The methodology compares conventional application-stack implementations with native XML and data-processing technologies, focusing on both runtime performance and development efficiency.

Scalability and performance are evaluated from two primary perspectives:

1. **Binding and data-processing acceleration**
   Evaluates different approaches for processing XML-based data, including object binding, streaming parsers, DOM-based processing, and XQuery. The use cases compare execution performance, resource consumption, scalability, and developer productivity across the different approaches.
2. **Mid-tier caching and service acceleration**
   Evaluates the impact of caching and data-storage strategies on service performance. The use cases compare native XML databases (XML DB) and relational databases (RDBMS) across different XML message sizes and database operations, including insert, update, delete, and query operations.

Last updated: October 15, 2026

Background and Goals

Software architects and developers choose XML parsing techniques, service libraries, encoding techniques, and protocols when building services using SOA techniques. Each choice has an impact on the scalability and performance of the finished service. In this example we have 3 goals:

1. Explain the changing landscape of APIs, libraries, encoding techniques, and protocols to software architects and developers. The current generation of technology choices changes approximately every 6 to 9 months. For instance, JAXB 1.0 is replaced by JAXB 2.0 and WebLogic Server 8.1 is replaced by WebLogic Server 9
2. Identify and use real-world, test use case scenarios showing software architects and developers how to choose technology based on their service goals
3. Deliver code compatible with the current techniques for building functional and scalability tests (black-box, unit, agile test-first). Many vendor tutorials are not compatible with what was used to build Appvance. While these packages could be modified with a lot of work, a better resolution was to develop kits with commercial, incorporating public information accessible to anyone

By pursuing these techniques we build a reusable method for evaluating Internet application performance and system scalability, plus the results feed basic business needs of cost/benefit and feature/function analysis including:

- Reduced hardware costs and reduced per-CPU licensed software by running more efficient services built with more efficient tools.
- Increased efficiency with lower network and processor bandwidth.
- Reduced time-solving interoperability problems with more effective tools.

The kit arms business managers and software developers with the evidence needed to recommend and adopt new solutions internally, and get their projects funded.

Last updated: February 22, 2023

If you have any feedback on this topic, please contact us at documentation@appvance.ai

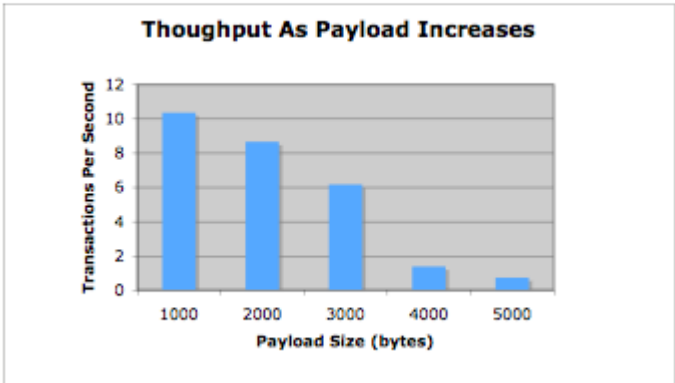
Add Calibration Testing to the Performance Test Plan

When defining the test scenario a set of assumptions is calculated into the Test Plan. For instance, the test scenario in The Test Scenario speculates a test case can achieve satisfactory throughput (in TPS) with a message payload request size of 500,000 bytes and 200 concurrent requests. A Calibration Test identifies a service agent's optimum throughput – measured in TPS at the consumer – against the given testing hardware and software. For instance, consider the data in Payload Size, Concurrent Agents, and Transactions -Per-Second Results.

Payload Size (Bytes)	Concurrent Agents	Transactions Per Second (TPS)
1000	10	10.376
2000	10	8.667
3000	10	6.174
4000	10	1.383
5000	10	0.731

Payload Size, Concurrent Agents, and Transactions-Per-Second Results list the two input values for the test: the message size sent to the service, and the number of concurrent test agents. Using these values, XSTest operates a test case by instantiating one thread for each concurrent request. Each thread dynamically generates the defined payload size data and sends it to the service as a request. Then the thread receives the server's response, validates the response, handles any exceptions, and logs the response as a completed transaction. The thread repeats the same steps until the test case period is finished. A Bar Chart of the Results Showing the Maximum Throughput Values displays the results from Payload Size, Concurrent Agents, and Transactions -Per-Second Results in a bar chart clearly shows the maximum throughput values.

A Bar Chart of the Results Showing the Maximum Throughput Values



Payload Size, Concurrent Agents, and Transactions -Per-Second Results and Scalability Index provide information about the service under test, including:

- As payload increases, TPS reduces proportionately. The test is not saturating or under-utilizing the server, network, or consumer. If TPS increases, the testing level was not high enough, or if it was flat or dropping sharply, the testing level was not low enough.
- The reduction in TPS is not proportional to the increase in request size. When the network and consumer are not at high enough activity levels, the service has a poor-performing request processor. One reason this could happen is if the message parsing system is not allocating resources (memory, network socket connections, or message queues) of the correct size for the demands of the test.
- TPS takes a significantly larger reduction for test cases above 3000 bytes of payload. In this case, a code profiler is used to find a test experiencing a buffer overflow or an undersized object list.

While A Bar Chart of the Results Showing the Maximum Throughput Values identifies a few things about the service, additional information is needed to conclude. Values, listed in Parameter Values Required to Calibrate a Test, are needed for the test parameters to conclude.

Parameter Values Required to Calibrate a Test

Parameter	Description
Request Payload Size	Service request message-body size (in bytes)
Response Payload Size	Service response message-body size (in bytes)
Concurrent Requests	Total number of concurrent requests
Transactions Per Second (TPS)	Ratio of total completed responses to execution time (in seconds)
Network Utilization	% network bandwidth (measured from server)
Server CPU Utilization	% server processor bandwidth
Consumer CPU Utilization	% consumer / client processor bandwidth
Average Transaction Time	Average service response time (measured by consumer / client)
Minimum Transaction Time	Minimum service response time (measured by consumer / client)
Maximum Transaction Time	Maximum service response time (measured by consumer / client)

In a stateless system, for each request, a service allocates its memory, CPU bandwidth, network bandwidth, and other resources needed to generate a response. For a stateless calibration test, resource bottlenecks are identified. Network and CPU Utilization shows the test scenario results including network utilization and server and consumer CPU utilization values.

Network and CPU Utilization

Parameter	Description	Transactions Per Second - TPS	Network Utilization	Server CPU Utilization	Consumer CPU Utilization
1000	10	10.376	1.24%	55%	34%
2000	10	8.667	1.14%	78%	37%
3000	10	6.174	1.32%	89%	31%
4000	10	1.383	0.45%	95%	21%
5000	10	0.731	0.28%	96%	18%

The results in Network and CPU Utilization give some idea of what is going on during the test scenario:

1. The test is server-bound preventing greater throughput (TPS). When payload sizes are less than 4000 bytes, server CPU utilization is high but not saturated. At 4000 bytes and greater, the CPU is saturated
2. Stateless tests require resources to handle the concurrent requests load. Take away a resource – CPU bandwidth or free memory – to operate on larger payloads and response times increase lowering overall TPS
3. The scale of the problem indicates there is a significant problem with the server. The payload size from 1000 to 5000 increases by a factor of 5, but TPS values decrease by a factor of 14, from 10.376 to 0.73In a stateless test, the TPS value should be proportional to the input

Since this is a stateless test, each request should be served from an independent group of resources (threads, memory, etc.). Watching CPU and memory utilization levels is an appropriate way to identify scalability and performance thresholds.

However, this is not the case for stateful services such as database and workflow applications. stateful services use data caches, and server queues and typically have session managers overhead. These items impact service CPU and memory utilization levels independent of the consumer request load.

For a scalability and performance test running in a defined software and hardware environment, a calibration test helps determine the appropriate service concurrent request levels and message payload sizes. The results show a Scalability Index for the service.

Last updated: March 18, 2024

If you have any feedback on this topic, please contact us at documentation@appvance.ai

Calibration What-if Chart

Calibration testing helps identify how a system behaves as the load level changes and provides a basis for selecting appropriate test levels. However, a calibration test cannot account for every possible system behavior or bottleneck.

The **Calibration What-If Chart** provides a set of common test observations, their likely causes, and recommended next steps. Use these observations to determine whether additional calibration is required before proceeding with the full performance test.

Test Experience	Likely Problem	What to Do Next
Increase in CRs with a decrease in TPS	Check the average response time.	Run a test case comparing response times as CRs increase. Identify the maximum acceptable response time and use it to determine the appropriate test level.
Increase in CRs with an increase in TPS	The test may not be calibrated for sufficiently high CRs or payload sizes.	Run additional calibration tests to determine the appropriate Scalability Index and corresponding CR and payload-size levels.
Increase in CRs with little change in TPS	CR levels may already be too high. Check CPU utilization if performing a stateless test.	Run another test with the CR level reduced, for example, by 50%, and compare the results.
Server CPU reaches approximately 95% utilization and increasing CR levels increases TPS	The test is likely exercising a stateful system behavior.	Run a test case to determine the service-under-test Scalability Index.
Consumer CPU reaches approximately 95% utilization and increasing CR levels decreases TPS	CR levels may be too high for the number of load-generating consumers.	Add more load-generating consumers and repeat the calibration test.
Consumer CPU utilization is approximately 15%, server CPU utilization is approximately 30%, and increasing CR levels produces little change in TPS	The network may be saturated or otherwise limiting throughput.	Check network bandwidth utilization. Consider adding network adapters to the server or using a faster network configuration.

Calibration testing provides the information needed to identify practical TPS levels for the system under test based on its current hardware, software, network, and workload configuration.

After calibration, use the resulting test levels to run the **full performance test** and evaluate the system under the intended workload.

> **Note:** The observations in this chart are guidelines rather than definitive diagnoses. Actual bottlenecks may vary depending on the application architecture, workload, payload size, network configuration, and test environment.

Last updated: August 14, 2026

If you have any feedback on this topic, please contact us at documentation@appvance.ai.

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:

```
Test Designer
      |
      v
Browser UX workflow
      |
      v
Instant Replay
      |
      v
Captured network traffic
      |
      +----> HAR
      +----> CSV
      +----> JS
```

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

For example:

```
Login
   |
   v
Search Product
   |
   v
Add Product
   |
```

```
    v
Checkout
```

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:

```
Concurrency
     ↓
Throughput / TPS
     ↓
Response Time
     ↓
Error Rate
     ↓
Resource Utilization
```

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
```

```
          |
          v
5. Create repeatable performance scenarios
          |
          v
6. Execute multiple controlled test runs
          |
          v
7. Collect performance and resource metrics
          |
          v
8. Analyze throughput, latency, errors,
   concurrency, and resource utilization
          |
          v
9. Identify scalability and capacity limits
          |
          v
10. Compare against baselines
          |
          v
11. Produce actionable conclusions
```

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.

Identify Test Environment Hardware and Software

Recommended guidance

Document the complete test environment before running scalability or performance tests. Results are only useful when the hardware, software, network, data, monitoring, and AIQ execution configuration are known and repeatable.

Capture these details

- Application servers, databases, caches, queues, load balancers, and external dependencies.
- CPU, memory, disk, network, operating system, runtime, browser, and driver versions.
- AIQ version, Test Node count, node sizing, browser mode, and execution geography.
- Data volume, account pool, DPL/CSV sources, and cleanup/reset process.
- Monitoring tools, metrics, log locations, and dashboard links.

Why it matters

Environment changes can explain different response times, error rates, and scalability limits. Record changes between calibration, baseline, stress, and final runs so results can be compared fairly.

Last updated: October 5, 2026

Method for Black Box and White Box Tests

Testing services for scalability and performance differs from testing individual software components or application code. Service performance testing focuses on understanding how a service behaves as the workload increases, including changes in concurrent requests, message sizes, message complexity, and response handling.

The Appvance methodology combines **black-box** and **white-box** testing. Black-box testing establishes how the service performs from the perspective of its consumer, while white-box techniques can then be used to identify the internal operations responsible for performance bottlenecks.

The recommended approach is:

1. **Establish a baseline using black-box performance testing.**
   Create a baseline performance metric, or **Scalability Index**, by measuring Transaction Per Second (TPS) at the service consumer. Test the service using different message sizes, message schema complexities, and levels of concurrent requests.
2. **Calibrate and compare the test environment.**
   When comparing multiple servers, consumers, brokers, or other service configurations, identify the **Performance Index** for each configuration. Normalize the test parameters and environment so that differences in infrastructure or workload do not produce misleading performance results. This calibration ensures that subsequent comparisons are meaningful.
3. **Identify internal performance bottlenecks.**
   After establishing the Performance Index, use white-box techniques such as code profiling to identify the operations that consume the most processing time when handling requests. Use this information to determine where optimization can provide the greatest performance improvement.
4. **Optimize and repeat the measurements.**
   Apply the identified optimizations and repeat the performance tests. Compare the new Performance Index with the previous baseline to determine whether the changes improved scalability or performance.
5. **Validate the final performance.**
   Run the Performance Index again under the calibrated test conditions and analyze the results. The final measurements provide an objective basis for evaluating the effectiveness of the optimizations and the scalability of the service.

This combination of black-box performance measurements and targeted white-box analysis provides the basis for the Appvance methodology for testing services for scalability and performance.

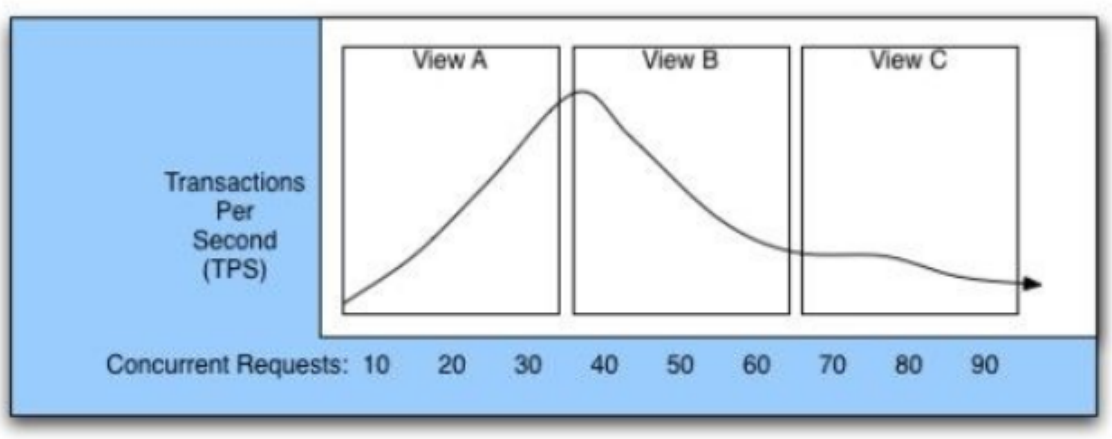
Last updated: August 12, 2026

If you have feedback about this topic, please contact documentation@appvance.ai.

Scalability Index

The **Scalability Index** describes how a service responds to increasing concurrent request (CR) levels and payload sizes. It is used during calibration to identify the range in which the service operates efficiently and to determine appropriate load levels for the Full Test.

The chart below illustrates three representative regions of a scalability curve:



- **View A — Underdriving:** Increasing the CR level results in a corresponding increase in TPS. The service is still able to process additional load efficiently, indicating that the current load level may be below the service's optimal operating range.
- **View B — Optimal operating range:** The service reaches its most useful operating range. Increasing the CR level no longer produces proportional throughput gains, but the service continues to operate at an acceptable TPS level. This range is typically used to define the CR levels for the Full Test.
- **View C — Overdriving:** Increasing the CR level produces little or no additional TPS. The service may be saturated, or another system resource may have become the limiting factor. Increasing the load further is therefore unlikely to provide useful scalability information.

The goal of calibration is to identify the **View B range** rather than simply maximize the CR level. Testing at too few requests may underutilize the service, while testing at excessive levels may introduce saturation that obscures the service's useful scalability characteristics.

Once the appropriate CR range has been identified, use those levels in the **Full Test**. The same calibrated levels can also be used when comparing alternative APIs, implementations, or products so that their performance results can be evaluated under equivalent workloads.

Identifying Bottlenecks

Resource utilization can help determine what limits scalability during calibration. Depending on the type of service and test, review metrics such as:

- CPU utilization
- Memory utilization
- Network bandwidth
- Load-generator or consumer utilization
- Service response time and TPS

The relevance of each metric depends on the architecture and behavior of the service. For example, CPU and memory utilization can provide useful information for stateless services, while resource utilization may be less representative of scalability for stateful services where other factors determine the system's behavior.

The **Scalability Index should therefore be considered together with system-resource metrics and response-time behavior**, rather than relying on TPS alone.

Last updated: August 13, 2026

If you have any feedback on this topic, please contact us at documentation@appvance.ai.

Scalability and Performance Kit Contents

Content	Description
Source Code	Complete source code for each use case and test scenario including Ant build scripts to build the kit in a preferred environment.
Developers Journal	A Developer's Journal describing in detail: Detailed use cases and test scenarios Design decisions and trade-offs XML and JAVA binding implementation stories Client-side software calling the implemented services Server-side software implementing the services Use case scenario specific findings Installation and performance tuning.
Pre-built JARs (start using immediately)	Pre-built JAR and WAR files for immediate use in any environment.
Appvance IQ and TestScenario Scripts	Scripts to stage a scalability and performance test of each use case and the test scenario.

Definitions: for Use Case and Test Scenario

Use the Appvance methodology to measure soap binding performance and scalability of bindings created and deployed using J2EE-based tools and XQuery and native XML database. Performance testing compares several methods to receive a soap-based Web Service request and responds to it. Scalability testing looks at the operation of service as the number of concurrent requests increases. Performance and scalability tests measure throughput as TPS at the service consumer.

The use cases and test scenarios contrast the TPS differences between the two most popular approaches to parse XML in a request, based on the following experiences:

1. A standard service test method for Web Services has not emerged. For instance, the SPECjAppServer1 test implements a 4-tier Web browser-based application where a browser connects to Web, application, and database servers in series. Internet applications, on the other hand, are truly a multi-tier architecture where each tier can make multiple soap requests to multiple services and data sources at any time. SPECjAppServer and similar 4-tier tests do not provide reliable Internet application information needed by capacity planners and software architects.

2. Software architects and developers specialize in service types. For instance, one developer works with complicated XML schemas in order processing services while another concentrates on building content management and publication services in portals
3. The tools, technologies, and libraries available for software architects change rapidly

Responding to these issues, the kit uses cases common to many SOA environments. These use cases highlight different aspects of SOA creation and present different challenges to the software development tools examined.

1. Compiled XML binding using BOD schemas. In this scenario, codenamed TV Dinner, a developer needs to code a part ordering service. The service uses Software Technology in Automotive Retailing (STAR) Business Object Document (BOD) schemas
2. On the consumer side, the test code instantiates a previously serialized Get Purchase Order (GPO) request document and adds a predetermined number of part elements to the ordered part. On the service side, the service examines only specific elements within the GPO instead of looking through the entire document
3. The developer's code addresses compartments by their namespace so they add/put only the changing parts of the purchase order. The other compartments (company name, shipping information, etc.) do not change from one GPO request to another. To accomplish this, the TV Dinner uses JAXB-created bindings allowing access to the individual compartments. This XML to object binding framework is used so only the required objects are instantiated
4. The TV Dinner scenario is named because, in a TV dinner, the entire dinner is delivered at once while the food is in compartments
5. Streaming XML (StAX) Parser. In this scenario, codenamed Sushi Boats, a developer builds a portal receiving a "blog" style news stream. Each request includes a set of elements containing blog entries. The test code scenario parameters determine the number of blog entries included in each request. The developer needs to take action on the entries of interest and ignore the others. The test code for the Sushi Boats features the JSR 173 Streaming XML (StAX) parser
6. The Sushi boat scenario is named from observations at a Japanese Sushi Bar where the food passes by in a stream and the diner selects the food they take from a selection of boats
7. DOM Approach. In this scenario, codenamed Buffet, a developer writes an order validation service receiving order requests and must read all the elements in a request to determine its response. The test code scenario parameters determine the number of elements inserted into each request. The test code for the Buffet scenario uses Xerces DOM APIs
8. The Buffet JAVAscenario is named from experience eating at a buffet restaurant and feeling compelled to visit all the stations

In addition to these use cases, the kit contrasts database performance differences between native XML databases and relational databases and stores XML data containing complex schemas and multiple message sizes in the mid-tier. The kit implements these use cases using both JAVA and XQuery tools.

Defining the Test Scenario

The Test Scenario is the aggregate of all use and test cases. For instance, the kit implements several use cases showing different approaches to XML parsing (DOM, XQuery, StAX, and the Binding Compiler). Therefore, running four use cases with two message sizes results in eight test cases in the test scenario. The Test Scenario lists the four use cases: 2 technology choices, 3 message payload sizes, and 4 concurrent request levels.

The Test Scenario

Use Case	Technology Choice	Request Payload Size	Concurrent Requests
JAVA XML Binding Compiler	XQuery Engine	5,000 bytes	5
JAVA Streaming XML Parser	JAVA Application Server	100,000 bytes	50
JAVA DOM		500,000 bytes	100
XQuery Streaming XML Parser			200

The test scenario is the aggregate of all the test cases. For instance, one test case uses the XML Binding Compiler running on an XML database at 100,000 bytes and 50 concurrent requests. This test scenario requires 96 test cases to run, given these parameters.

If each test case has a 5-minute warm-up period, takes 5 minutes to run, and has a 5-minute cool-down period, the test scenario requires 1,440 minutes (24 hours) to run. Seeing how run time can increase significantly as the number of use cases increases, use caution when adding use cases to a test scenario; add the use cases if the resulting actionable knowledge is necessary.

Last updated: March 18, 2024

If you have any feedback on this topic, please contact us at documentation@appvance.ai

Appvance Method

Phase	Goal	Checkpoint
Planning	Answer the question: How will this test benefit the organization?	Write Test Plan document.
Definition	Identify use cases, test cases and scenario, and test environment (hardware, software and network)	Add the use cases, test cases and test scenario to the Test Plan.
Calibration Test	Calibrate the test cases to the test environment.	Identify the use cases driving the test environment to its maximum throughput (as measured in TPS from the client).
Optimize	Modify the service and / or test environment to optimize for the best performance based on what was learned in the Calibration Test.	Amend the Test Plan to add the changes for optimization.
Full Test	Run the *TestScenario*	Successful run of *TestScenario* .
Results Analysis	Identify test result metrics and trends against the *TestScenario* goals.	Present results and achieve adoption by management.

Here is a brief explanation of the terminology used in The Appvance Test Methodology

- Use Case – describes the functionality of a test. For instance, a test comparing XML parsing techniques includes two use cases: the first uses the Xerxes DOM parser and the second uses the JAXB XML binding compiler
- Test Case – describes the inputs to an individual test. For instance, for a test comparing service throughput at low and high levels for message payloads, define 2 test cases: the first makes requests to the service using an XML document of 500 bytes and the second sends XML documents of 10,000 bytes
- Test Scenario – the aggregate of all test and use cases to run a complete test

There is, however, an important distinction between this SOA test method and typical software testing.

Last updated: June 02, 2023

If you have any feedback on this topic, please contact us at documentation@appvance.ai

Understanding the Tests

The Appvance Test Methodology defines common test goals used when testing Internet application services. Because application architectures and service technologies continue to evolve, the methodology is not limited to these test goals.

The following concepts are used to understand the tests defined in the methodology:

- **Test Name** identifies the test being performed.
- **Test Benefit** describes the primary benefit or objective of the test.
- **Test Type** determines whether the test is **stateful** or **stateless**. Each type requires a different testing strategy and addresses different performance goals.
- **Scalability Index Parameters** define the test inputs used to measure the Scalability Index. These inputs represent expected usage patterns and help determine how service performance is expected to behave under production-like conditions.

Test Name	Test Benefit	Test Type	Parameters Related to the Scalability Index
Service Interface	Decrease time for service request responses to lower network bandwidth and server hardware costs.	Stateless	Message size and concurrent request level.
XML Parsing	Decrease time for routing service messages to lower network bandwidth and server hardware costs.	Stateless	Schema complexity (depth and element count), document size and concurrent request level.
Data Persistence	Decrease time for storing and retrieving messages to lower network bandwidth, server hardware and disk costs.	Stateful	Schema complexity (depth and element count), document size, concurrent request level.
Data Transformation	Decrease time for transforming a message into a given XML schema to lower network bandwidth and server hardware costs.	Stateless	Source and destination schema complexity (depth and element count), request and output document size, concurrent request level.
Data Aggregation and Federation	Decrease time for responding to service requests requiring up-stream data to reduce network bandwidth and server hardware costs.	Stateful	Schema complexity (depth and element count) for up-stream services, data persistence quantities, message time-to-live (TTL) values.
Data Mitigation	Reduce time when a service is unavailable at peak usage to improve service availability and user satisfaction.	Stateful	Schema complexity (depth and element count) for each request, document size, concurrent request level.

Test Types

Stateless Testing

Stateless testing is used for services that process each request independently of other requests. The test evaluates how the service responds as factors such as **concurrent requests** and **message payload size** change.

Stateful Testing

Stateful testing is used for services that maintain state or depend on information from previous requests. This can include services that provide:

- Data persistence
- Workflow or transaction processing
- Message queuing
- User sessions
- Data indexing

Stateful testing evaluates how these state-dependent operations affect service performance and scalability under different workloads.

Scalability Index Parameters

The parameters associated with the **Scalability Index** define the inputs used during the test, such as concurrency levels and payload sizes.

These parameters represent the expected usage pattern of the service in production and are used to determine how service performance changes as the workload increases.

The Appvance Test Methodology defines the series of steps required to plan, execute, and analyze these tests.

For more information, see Appvance Method

Last updated: August 19, 2026

If you have any feedback on this topic, please contact us at documentation@appvance.ai

Understand TPS in AIQ Performance Testing

Recommended guidance

Transactions per second (TPS) measures how many completed business transactions or requests are processed during a test window. In AIQ performance testing, define TPS carefully before comparing runs because different teams may count transactions, pages, API calls, or user journeys differently.

What to define

- The exact transaction boundary: request, step, business action, or full journey.
- The measurement window used for average, peak, and sustained TPS.
- Whether failed, timed-out, retried, or partially completed transactions are included.
- The user load, ramp-up, pacing, test data, and environment used for the result.
- The related metrics: response time, error rate, throughput, CPU, memory, and network utilization.

How to use TPS

1. Validate the script and data at low load before treating TPS as meaningful.
2. Compare TPS only between runs with the same scenario, load model, and environment.
3. Watch for increasing error rate or response time when TPS rises.
4. Document whether the reported TPS is target, achieved, peak, or sustained.

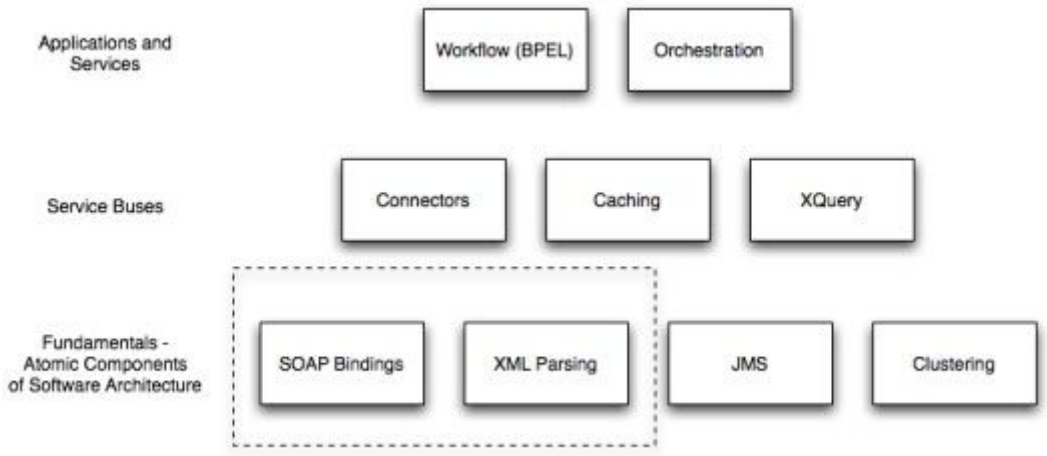
Last updated: October 6, 2026

User Goal Oriented Testing (UGOT)

Countering software developers by saying "If you design for every possible user, no individual user will have their goals met when they use your software application!" Coverage tests are typically pointless, as users typically take a path through the functions in service, one after another, like chain links. The agile development community approaches this problem by recommending a test-first strategy, which is to write a unit test of a class before writing the class itself. At build-time, the compiler environment compiles the object code for the class and then runs the unit test against the compiled code. The unit test completes successfully by receiving example data and validating the response. If the class returns an invalid response, the unit test throws an exception that the build and deploy environment handles.

Unit testing and agile development methods help but are not a complete solution to UGOT techniques. For instance, a test first is usually only carried out on a unit level. SOA deploys applications as a collection of services, so testing individual units miss most of the big problems that occur during SOA integration and deployment. UGOT modeled tests check service as an individual user would – by picking one feature after the next in a chain of service requests, which is a more global and inclusive testing mechanism.

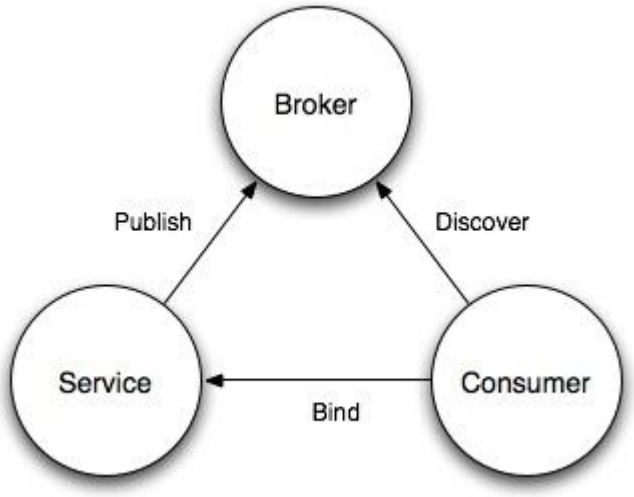
UGOT is ideal to understand performance and scalability. Internet applications must deliver real value to those who build and operate services. Understanding what is being tested is the heart of the issue, as can be seen in the following chart that shows the building blocks of Java development tools for building applications.



The components build on each other in three tiers:

- At the bottom are the fundamental components for soap bindings, XML parsing, JAVA inter-application messaging services (JMS), and clustering
- Building on these is service bus components for services to inter-operate at a message level
- The top tier provides interoperability at the application level

Given these building blocks and tiers, testing should be done across the tiers, where key performance bottlenecks in soap Bindings and JMS services may happen. Each of the components in the above chart impacts the scalability and performance of application implementation. Additionally, understanding the testing goal can be seen by reviewing the definition of service architecture as shown in the following illustration.



Which should be tested and when should it be tested? Performance tests normally check at least two. For instance, one test may check a consumer and service, and another check a consumer and a broker.

The Appvance Methodology lists options to understand what part of the system would benefit most from scalability and performance analysis.

Last updated: October 15, 2026

If you have any feedback on this topic, please contact us at documentation@appvance.ai

Applying UGOT

Define the user goal, the business outcome, and the sequence of actions that a representative user must complete. Use those goal-oriented flows to prioritize coverage and performance measurements, while retaining unit and component tests for lower-level defects.

Performance Testing Primer for AIQ

Your quick guide to achieving the best test results without reading entire books on Performance Testing

What is Performance Testing?

We ultimately want to understand the performance of the application or system under various conditions. This requires a number of specialized tests to understand the behavior and guarantee stakeholders that the system will perform well under reasonable circumstances. Performing well includes transaction times as well as not crashing.

Why is it important?

People are used to 300ms performance from Google and sub-1-second performance from Amazon. The world has been conditioned to very fast response times. People begin to get distracted after 1 second, and at 7 seconds 50% have moved on at least mentally to another task. Performance testing at each and every build catch issues that can crop up at any time.

Types of Tests

- **Performance** | What is the transactional performance of my application to the user (through the UX) under a variety of conditions from 1 user to N users?
- **Load** | How do my services perform under load at the API level?
- **Stress** | What happens when my application is under stress for a short while or longer?
- **Scalability** | How does my application scale?

- **Soak** | How does my application perform over time specifically concerning memory leaks or other behavior which changes over time?
- **Spike** | Does my application recover after a large spike in traffic?
- **Database (direct)** | How do direct database interactions impact throughput and performance?

Testing in Today's DevOps World

Performance testing should be a standard part of every build where you build, test, tune, and start again. Performance improvements occur, and catching performance issues before your users do because it happens at each and every build. Correcting issues in build 144 that occurred in build 72 is not fun, nor productive.

Create Repeatable Real-User Simulation

Tests must be **realistic**. That is, closely resembles the way real users will interact with the system as well as repeatable so that you can track progress.

1. Identify the maximum number of users the system will ever see (call that Max Design Capacity)
2. Identify reasonable SLAs (max transaction times) that your users should tolerate
   A modern target would be under 1 second response times through the UX
   A good start would be under 3 seconds for all transactions and work towards 1 second
3. Create a mix of relevant use-cases at the UX level as well as optionally at the API level and database level
4. A good scenario will have many use cases represented which interact with the system in a realistic way and activate all aspects of the system such as the database and other layers
5. Data-drive those transactions for more realistic scenarios
6. Randomize think times and interval times to create more realistic scenarios
   Generally, randomize think times between 1 and 10 seconds
   Generally, randomize interval times between 5 and 20 seconds

Setup and Run Tests

A. Monitor

Place APCMonitor on your web, application, database, and all other servers associated with the app under test.

B. Measure Transaction Times Without Load

Run a single user performance test with one scenario repeated 10x in series. This both ensures that everything is functioning correctly including your application and that the nodes can see the application. This will also provide a baseline for transaction times

1. You should not see any errors with this 1-user test. If you see a consistent error you may have a script or application issue.
2. You may see some steps which take a long time (over ten seconds) in the report. You can validate these manually and see what actually happens. Some steps can have a lot of elements loading besides the visible ones and the browser will still show in an active state, and Appvance will await all elements to come in, even though a real user may have moved on. So you may decide to ignore this step results for the moment as it won't impact a user, but more that there could be better ways to code that Ajax activity in the future. Also, this can be different for each browser.
3. If you do not achieve your SLAs for transaction times at this step, STOP and address the issues before proceeding.

C. Calibrate the Test Nodes

It is critical to calibrate a test node for max capacity for your exact Scenario. Each scenario requires new calibration.

Follow the calibration guidelines which include ramping up users on one test node until that node reaches CPU and memory limits, or other aberrations occur. When other issues occur these could be due to your application being at its limit (and **not** the test node) or could occur because GPU or other non-monitored data are at a limit on the test node. Back down by 20% from any limit and call that the calibrated maximum per node. Enter that number into each test node setting in the scenario going forward and assign enough nodes to get to the totals you need.

D. Run a Scalability Test

Create a scalability test by picking at least 3 levels of load (it can be more) and setting that up to automatically occur within one Scenario.

This can provide performance, load, scalability, and stress all in one thoughtfully designed test. The top-level should be at or above the design maximum for the application. For example, My application is designed for 900 concurrent users. I would set up a scalability test for 100, 300, 600, 900, and 1200 users to run that sequence automatically.

1. Run each level of load for enough time that each use case could be re-run 5-10x. So an average 2-minute use case run 5x would need at least 10 minutes at each load. This ensures more data for average
2. Ignore error rates below 10%. Some transactions may fail for a variety of reasons but since this is not a functional test, we are only interested in the transaction times and scalability
3. Transaction failures may approach 100% when you have gone beyond the design maximum of the application
4. The key chart to review at the end of the test is the Scalability Index. You will want linear scalability. Anything not linear shows your system is not keeping up with requests
5. You will want to investigate the full CSV data file and look for transaction times that exceed the SLA
6. No app-related servers should get above 90% resource usage on any resources such as CPU, memory, HEAP, etc. as reported by APCMonitor. Should any servers see levels this high, additional hardware is required for further scalability.

E. Run a Soak Test

It is important to run a soak test for at least 24 hours to assess any abnormalities which occur over time. Use the same scenarios as above and pick a mid-level of load which had few errors and was within the linearly scalable range of the application. Setup the Scenario to ramp up over an hour and stay at the chosen level of load for 24 hours and then ramp down.

F. Run a Spike Test

A spike test can assess how your systems recover after a fast spike in users. You can set this up using the same scenario as above. Change it to run for 10 minutes at 20% of max design capacity, and then spike for 5 minutes to 150% of design capacity and then drop back to 20% for another 20 minutes. The system should recover properly and meet SLAs after the spike.

Troubleshooting

Possible Database Issues

- Insufficient indexing: Tune database indexing to improve query processing
- Fragmented databases: Place table records on adjacent pages
- Out-of-date statistics: Degrade query optimizer performance
- Faulty application design: Excessive DB calls, excessive data requests
- Many other important database issues can be found here:

  https://dzone.com/articles/reasons-slow-database

Typical Web and App Server Issues

- Poor server design: Inefficient data or page caching
- Memory problems: Physical memory constraints
- High CPU usage: Usage >80% indicates problems
- Poor database tuning: The application server sending too many DB requests
- Poor cache management: Produces high CPU usage, disk access
- Poor session management: Produces high CPU usage, disk access, Time-outs
- Poor security design: Excessive use of HTTPS protocol
- Many other important issues can be found here:

  https://www.serverwatch.com/servers/server-optimization/

Network Issues

- Firewall capability
- Internet access throughput, packet loss, delays, and bandwidth
- Load balancers, gateways, routers
- Many other important issues can be found here:
- http://www.dataexpedition.com/support/notes/tn0009.html

Resolving the Issues

Depending on time, expertise, and accessibility, one might choose different items to address first. Given the very low cost of CPU cores and memory today, it's hard to justify spending manpower on addressing issues before spending $$ on faster hardware.

- Upgrade hardware: RAM, CPU, network bandwidth (RAM and CPU cores are cheap!)

- Improve current application design: Algorithms, caching, DB calls, memory usage
- Upgrade software infrastructure: OS, web server, database
- Upgrade system architecture: Client-server to basic n-tier, basic n-tier to enterprise n-tier, software and hardware changes

---

Last updated: July 03, 2023

If you have any feedback on this topic, please contact us at documentation@appvance.ai

Test Designer Performance Scalability: Maximizing the number of concurrent browsers per node

For performance testing, you can maximize the number of concurrent browsers that can be opened in a node.

For AWS EC2 nodes running AIQ, the number of concurrent Chrome browsers that can be assigned per node is around 20 to 30, depending on the application.

The main bottleneck to achieving more concurrent browsers is the speed of the storage volume as measured in IOPS (input/output operations per second). A typical 60GB EBS volume is not rated higher than 3000 IOPS.

To avoid the bottleneck of volume speed, you can leverage **AWS EC2 instances that include NVME SSD volumes**. These types of volumes are very fast, but they are ephemeral storage volumes. Any data stored in these volumes will be lost once the instance is stopped or terminated. Also, the data in the NVME SSD volumes are not saved in the AMI images.

So the process of starting appvance in one of this nodes involves several initial steps which are coded in the provided **nvme_setup.sh** script recommended for performance nodes. The script does:

1. Mount the NVME SSD storage as a device visible to the OS
2. Copy GraalVM and aiq_distribution folder to the NVME SSD
3. Modify global.json in aiq_distribution/NetworkProxy/config/global.json so Chrome browser has the following option added:

   ```
   {     "name": "experimentalDS3",     "value": "--disk-cache-dir=cache||--disk-cache-size=1"      }
   ```

4. Start AIQ from the NVME SSD drive.

Setting up a Amazon Linux 2 node for performance testing

1. Add the following file (nvme_setup.sh) to the aiq_distribution folder in the Amazon Linux 2 instance.
   This script will mount the NVME SSD volume, if found, and then copies the Graalvm and aiq_distribution folders to the SSD drive.
   Note: This script assumes the AIQ is located at /opt/aiq/aiq_distribuition. Edit the script if the AIQ location is different.
   nvme_setup.sh

```
#!/bin/bash# This script assumes:#  - using a AWS ec2 instance with an ephemeral SSD NVME volume (for example m5ad, c5d instance families)#  - AIQ is installed at /opt/aiq/aiq_distribution, modify below if not.#  - Linux has NVME volumne drivers (included in Amazon Linux 2 OS)#  - graalvm is installed and $JAVA_HOME is set to its locationif [ -e '/dev/nvme1n1' ] ; then    echo "Mounting NVME volume as /data"    sudo file -s /dev/nvme1n1    sudo mkfs -t xfs /dev/nvme1n1    if [ ! -d '/data' ] ; then       sudo mkdir /data    fi    sudo mount /dev/nvme1n1 /data    sudo chown ec2-user:ec2-user /data    sleep 5    FOLDER=$(basename $JAVA_HOME)    sudo cp -R $JAVA_HOME /data    export JAVA_HOME=/data/${FOLDER}    export PATH=/data/${FOLDER}/bin:$PATH    echo "JAVA_HOME: $JAVA_HOME"    sudo cp -R /opt/aiq/aiq_distribution /data/aiq_distribution    sudo chown -R $USER:$USER /data    cd /data/aiq_distribution/TestNodeelse   echo "Skipping mounting /data as instance do not have an NVME volume (/dev/nvme1n1)"fibash startup.sh
```

2. Modify global.json in aiq_distribution/NetworkProxy/config/global.json so that Chrome browsers have following option added:

```
{      "name": "experimentalDS3",      "value": "--disk-cache-dir=cache||--disk-cache-size=1"      }
```

3. Start AIQ started using a systemd serviceDo not use /etc/rc.local.
The systemd service can set the maximum number of open files per process. This is needed in

linux distros because the default of 1024 files per process is too low.
Note: Staring AIQ as a service in this manner is not compatible with the Lab Management automatic upgrade for linux nodes.

To configure the service create a text file named appvance.service in the /etc/systemd/system folder with the following contents. Edit the paths in file as needed to match your configuration.

```
[Unit]Description=Start Appvance IQ
servicesAfter=syslog.target[Service]Environment=PATH=/opt/graalvm-ce-java11-
21.3.3.1/bin:/sbin:/bin:/usr/sbin:/usr/binEnvironment=JAVA_HOME=/opt/graalvm-ce-java11-
21.3.3.1User=ec2-userType=simpleLimitNOFILE=10000000LimitNPROC=20000ExecStart=/bin/bash
/opt/aiq/aiq_distribution/TestNode/nvme_setup.shWorkingDirectory=/opt/aiq/aiq_distribution/Tes
tNodeTimeoutStartSec=0[Install]WantedBy=multi-user.target
```

After creating above file, execute the following commands in the shell:

`systemctl daemon-reload`
`systemclt enable appvance`

With the above configuration, an instance with size of m6id.2xlarge AIQ was able to execute a test designer test case using 40 chrome browsers concurrently. An instance with a size of m6id.4xlarge was able to execute 80 chrome browsers concurrently. Please note that specific values might vary depending of the application under test.

Capacity and operational considerations

Validate the browser-per-node target with the actual application and node profile before selecting a production capacity. Monitor IOPS, CPU, memory, and browser stability during a representative run. NVMe instance storage is ephemeral; keep AIQ distributions and test assets in a reproducible source or image and rerun the setup script after a stop or replacement.

Last updated: October 15, 2026

If you have any feedback on this topic, please contact us at documentation@appvance.ai