Performance Testing Report

CS423 - Software Testing

Group 06
Lê Thanh Lâm - 22125046
Nguyễn Hoàng Phúc - 22125076
Nguyễn Chính Thông - 22125102
Nguyễn Võ Hoàng Thông - 22125103

Contents

1 · Introduction

1.1 · Overview

Performance testing is a type of **non-functional testing** focused on evaluating how a system behaves under specific workloads. Unlike functional testing, which verifies correctness, performance testing measures **responsiveness, stability, scalability, and resource usage**.

Key performance attributes include:
- **Response time** – How quickly a system responds to a request.
- **Throughput** – The number of transactions processed per second.
- **Resource utilization** – CPU, memory, disk, and network usage.
- **Scalability** – How well the system handles increasing load.

Performance testing ensures that software:

- Meets service-level agreements (SLAs).
- Can scale under expected user loads.
- Identifies performance bottlenecks before deployment.

1.2 · Importance

Performance testing is crucial for:
- **User satisfaction** – Slow systems degrade user experience.
- **System reliability** – Detects failures under high concurrency.
- **Cost optimization** – Avoids over-provisioning or inefficient scaling.
- **Business continuity** – Prevents outages during peak usage.

2 · Theoretical Background

2.1 · Key Concepts and Metrics

Term	Definition
Response Time (RT)	Time between request submission and response completion.
Throughput (TP)	Number of transactions or requests handled per second.
Latency (L)	Delay before a response begins.
Concurrency	Number of simultaneous users or sessions.
Error Rate	Percentage of failed transactions.
Resource Utilization	CPU, memory, network, or I/O usage during test execution.

2.1.1 · Relationship between metrics

The fundamental relationship between throughput and response time is:

$$\text{Throughput} = \frac{\text{Number of Requests}}{\text{Total Time}}$$

Response time directly influences perceived performance.

2.2 · Types of Performance Testing

Type	Purpose	Example Scenario
Load Testing	Measure system behavior under expected user load.	100 concurrent users accessing a website.
Stress Testing	Determine system limits by increasing load beyond capacity.	Simulate 10,000 users until failure.
Spike Testing	Evaluate system's recovery from sudden load spikes.	Sudden traffic surge during a product launch.
Endurance (Soak) Testing	Assess stability over long durations.	Continuous API calls for 24 hours.
Scalability Testing	Measure performance while scaling resources.	Increasing number of nodes in a cluster.

2.3 · Performance Testing Lifecycle

1. **Requirement Gathering** – Identify performance goals (e.g., "handle 500 concurrent users with < 2s response time").
2. **Planning** – Define workload model, test scenarios, and acceptance criteria.
3. **Design & Implementation** – Develop test scripts (e.g., with JMeter or k6).
4. **Execution** – Run tests in a controlled environment.
5. **Monitoring & Analysis** – Record system metrics and interpret results.
6. **Reporting & Optimization** – Identify bottlenecks and recommend improvements.

"Performance testing is iterative: each test cycle feeds insights into performance tuning and architecture refinement." — Jain, R. (1991)

2.4 · Workload Modeling

Workload modeling involves simulating real user behavior:
- Define **user profiles** (types of users or transactions).
- Establish **think time** between user actions.
- Distribute **requests per second (RPS)** and concurrency.

Mathematically, a simple workload can be modeled as:

$$W = N \times \left(\frac{R}{T}\right)$$

where **N** = number of users, **R** = requests per user, **T** = total test duration.

2.5 · Tools and Framework Context (Preview)

While tools are discussed later, understanding their role helps in theory:
- **Apache JMeter** – simulates HTTP requests, measures throughput and latency.
- **k6** – modern load-testing tool using JavaScript scripting.
- **Locust** – Python-based, user-oriented load testing.

3 · Performance Testing Methodology

3.1 · Step-by-Step Process

1. **Define Performance Goals**
   - Examples: "Average response time ≤ 2s", "Error rate ≤ 1%", "CPU ≤ 70% utilization".
   - Goals should be measurable and traceable to business needs.
2. **Identify Test Scenarios**
   - Select critical user flows (login, checkout, API endpoint).
   - Prioritize based on system usage frequency and risk.
3. **Design Test Scripts**
   - Simulate virtual users executing real workloads.
   - Parameterize requests (dynamic data, sessions).
   - Include assertions for success criteria.
4. **Set Up Test Environment**
   - Isolate performance environment from production.
   - Ensure representative hardware, network, and databases.
   - Use monitoring tools (Grafana, Prometheus, New Relic).
5. **Execute and Monitor Tests**
   - Gradually increase load to observe threshold and breaking points.
   - Collect metrics: latency, throughput, CPU/memory, network I/O.
   - Observe logs and error responses.
6. **Analyze and Report Results**
   - Compare metrics against baseline and goals.
   - Identify bottlenecks (e.g., slow DB queries, memory leaks).
   - Visualize results via charts (JMeter HTML report, Grafana dashboards).
7. **Optimization and Retesting**
   - Tune configurations (e.g., caching, DB indexing, thread pools).
   - Retest after each optimization to validate improvements.

3.2 · Common Performance Bottlenecks

- **Database issues**: inefficient queries, missing indexes.
- **Server-side logic**: blocking I/O, excessive thread contention.
- **Network latency**: bandwidth constraints, high round-trip times.

- **Frontend performance**: heavy assets, client-side rendering delays.

3.3 · Best Practices

- Test early and continuously (shift-left performance testing).
- Keep test data realistic and production-like.
- Correlate metrics with business KPIs.
- Automate performance tests in CI/CD pipelines.
- Use statistical methods (percentiles, averages, standard deviation) for analysis.

4 · Tools & Frameworks

Performance testing tools and frameworks have evolved significantly to meet diverse testing requirements. Understanding their characteristics and classifications helps teams select the most appropriate tools for their specific needs.

4.1 · Classification of Performance Testing Tools

Performance testing tools can be categorized along several dimensions, each offering different advantages and trade-offs:

4.1.1 · Interface Type: Script-based vs. UI-based

Type	Strengths	Weaknesses	Examples
Script-based	• Full flexibility and control • Version control friendly (Git) • Easy CI/CD integration • Reusable code components • Advanced customization • Better code review process	• Steeper learning curve • Requires programming skills • Slower initial setup • Less intuitive for beginners • Debugging can be complex	Locust, K6, Gatling, Artillery
UI-based	• Low barrier to entry • Quick test creation • Visual workflow design • Record and playback • Easy for non-programmers • Immediate visual feedback	• Less flexible for complex scenarios • Harder to version control • Limited code reusability • UI can be resource-intensive • May still need scripting for advanced cases	JMeter, LoadRunner, NeoLoad

4.1.2 · Deployment Model: Local vs. Cloud-based

Type	Strengths	Weaknesses	Examples
Local	• Full control over environment • No recurring cloud costs • Data stays on-premise (security)	• Limited by local resources • Cannot simulate global users easily • Infrastructure maintenance burden	JMeter, Locust, Gatling, K6 (OSS)

Type	Strengths	Weaknesses	Examples
	• No internet dependency • Predictable performance • One-time setup cost	• Scaling requires hardware investment • Single point of failure	
Cloud-based	• Massive scalability on-demand • Geographic distribution • No infrastructure management • Pay-as-you-go model • Rapid test execution • Built-in monitoring/reporting	• Recurring subscription costs • Data leaves premises (compliance issues) • Internet dependency • Vendor lock-in risks • Potential latency issues • Less control over infrastructure	BlazeMeter, K6 Cloud, Loader.io, AWS Device Farm
Hybrid	• Best of both worlds • Gradual cloud migration • Flexible resource allocation • Cost optimization options • Local dev, cloud production testing	• Complex setup and management • Requires expertise in both models • Potential sync issues • Higher overall complexity	K6, Gatling Enterprise, JMeter + Cloud

4.1.3 · Programming Language & Ecosystem

Language	Tools	Strengths	Weaknesses
Python	Locust, Molotov	Easy to learn, rich libraries, great for data processing, readable syntax, wide adoption	Slower execution, GIL limitations, higher memory usage
JavaScript	K6, Artillery, Playwright	Web-native, async/await support, large ecosystem, modern tooling, Node.js performance	Callback complexity, less suitable for CPU-intensive tasks, runtime quirks
Java	JMeter, Gatling	Enterprise-grade, robust, excellent performance, mature ecosystem, strong typing	Verbose syntax, slower startup, higher resource usage, steep learning curve
Go	Vegeta, Hey	Excellent performance, low resource usage, built-in concurrency, fast compilation	Smaller ecosystem, less libraries, simpler language features, smaller community
Ruby	Tsung	Rails integration, elegant syntax, developer-friendly, quick prototyping	Performance limitations, smaller performance testing community, less tooling

4.1.4 · Protocol Support

Category	Strengths	Weaknesses	Examples
Multi-protocol	• Single tool for diverse protocols • Comprehensive testing capability	• Complexity overhead • Steeper learning curve • May not excel at any single protocol	JMeter, LoadRunner

Category	Strengths	Weaknesses	Examples
	• Reusable infrastructure • Covers legacy systems • Standardized approach	• Larger footprint • Plugin dependency	
HTTP-focused	• Optimized for web/API • Modern features (HTTP/2, WebSocket) • Better performance for HTTP • Simpler, cleaner API • Faster execution	• Limited to HTTP-based protocols • Cannot test databases directly • Not suitable for legacy systems • May need multiple tools	K6, Locust, Artillery
Specialized	• Domain expertise • Highly optimized • Specific metrics • Deep protocol knowledge • Accurate simulation	• Limited scope • Additional tools needed • Smaller communities • Less general applicability • Niche use cases only	HammerDB, JMeter plugins

4.2 · Popular Performance Testing Tools

4.2.1 · Open Source Tools

Locust
- Python-based, script-driven
- Distributed load generation
- Real-time web UI for monitoring
- Event-driven, can simulate millions of users
- Best for: Teams comfortable with Python, API testing

K6
- JavaScript-based, modern developer experience
- Built-in metrics and thresholds
- Cloud and local execution
- Excellent CI/CD integration
- Best for: DevOps teams, modern web applications

JMeter
- Java-based, industry standard
- Both GUI and CLI modes
- Extensive protocol support and plugins
- Large community and resources
- Best for: Enterprise environments, complex protocols

Gatling
- Scala-based, code-as-configuration
- High-performance engine
- Detailed HTML reports
- Strong IDE support
- Best for: High-throughput testing, Scala/Java teams

Artillery
- JavaScript/Node.js based
- YAML configuration with JS extensions
- Serverless testing support
- Quick setup and execution
- Best for: Quick tests, microservices

4.2.2 · Commercial Tools

Tool	Key Features	Best For
LoadRunner	Enterprise-grade, comprehensive protocols, advanced analytics	Large enterprises, complex applications
NeoLoad	Codeless design, APM integration, dynamic infrastructure	Agile teams, SAP testing
BlazeMeter	JMeter-compatible, cloud-based, geo-distributed testing	Teams using JMeter wanting cloud scale
K6 Cloud	K6 script compatible, performance insights, team collaboration	K6 users needing cloud features

4.3 · Tool Selection Criteria

When selecting a performance testing tool, consider:

1. **Technical Requirements**: Protocols needed, application architecture, integration points
2. **Team Skills**: Programming language familiarity, learning curve
3. **Budget**: Open source vs. commercial licensing costs
4. **Scale**: Expected load levels, geographic distribution needs
5. **CI/CD Integration**: Automation capabilities, reporting formats
6. **Reporting & Analysis**: Visualization needs, stakeholder requirements
7. **Community & Support**: Documentation quality, available resources

4.4 · Modern Trends

The performance testing landscape is evolving with several key trends:

- **Shift-left Testing**: Earlier performance testing in development cycle
- **Cloud-native**: Kubernetes-aware testing, containerized test execution
- **Observability Integration**: Correlation with APM, distributed tracing
- **AI/ML**: Intelligent test generation, anomaly detection
- **Developer-centric**: Code-based tests, version control, modern languages
- **Continuous Performance**: Automated performance gates in CI/CD pipelines

5 · Tools & Setup - Apache JMeter

5.1 · What JMeter is

Apache JMeter is a 100% Java open-source application for load and performance testing. It simulates multiple users (threads) sending requests to a server and collects response metrics (latency, throughput, errors).

5.2 · Installing and Preparing JMeter

- **Prerequisite**: Java JDK (use the latest LTS supported by JMeter).
- **Download**: Official binary from the Apache JMeter site; unzip and run `bin/jmeter` (GUI) or `bin/jmeter.bat` on Windows.

Important JVM / OS recommendations (practical):

- Increase JMeter JVM heap if you plan many threads or large listeners (edit `JVM_ARGS` in `bin/jmeter` or `jmeter.bat`, e.g. `-Xms1g -Xmx4g`). The user manual and best-practices strongly encourage tuning the JVM for heavy tests.
- **Do not** run large load tests in GUI mode - use non-GUI mode for execution to avoid the GUI overhead. The manual and best-practices emphasize non-GUI execution for real load runs.

5.3 · Core JMeter Components

- **Test Plan** - root of a JMeter test suite.
- **Thread Group** - defines virtual users, ramp-up, loop count.

- **Samplers** - HTTP Request, JDBC Request, etc. (what actions virtual users perform).
- **Logic Controllers** - control flow (If Controller, Loop Controller, Transaction Controller).
- **Timers** - add think time (Uniform Random Timer, Constant Timer). Realistic workloads require think time.
- **Assertions** - verify responses (status code, JSON path, size). Useful to ensure validity under load.
- **Config Elements** - HTTP Header Manager, CSV Data Set config (parameterization), Cookie Manager.
- **Listeners** - View Results Tree, Aggregate Report, Summary Report - used for debugging and result export. For heavy runs, minimize GUI listeners and instead write results to disk for post-processing.

5.4 · JMeter Plugins and Extensions (Brief)

JMeter has a plugin ecosystem (e.g., JMeter Plugins Manager) that adds advanced samplers, timers, and graphs. Plugins are useful but always validate their stability for your test environment (stick to essential plugins for reproducibility).

5.5 · Distributed Testing (Scaling Load)

If a single machine cannot generate required load, JMeter supports **remote (distributed) testing** - one client (controller) drives multiple remote JMeter servers (agents) to multiply the virtual user capacity. Follow the official remote testing steps and RMI configuration; ensure network/firewall and identical test plan & plugin versions on all nodes.

6 · Practical Demonstration - Example Test Plan

6.1 · Example Test Scenario (Choose One for Demo)

Target: A RESTful API `https://api.example.com/v1/items` - typical CRUD flows.

User flows to test (prioritized):
1. Browse items (GET /v1/items?page=X) - 60% of traffic
2. Get item details (GET /v1/items/{id}) - 30%
3. Create item (POST /v1/items) - 10% (requires authentication + dynamic data)

6.2 · Test Plan Structure (High Level)

- **Test Plan**:
  - Thread Group (Virtual Users: configurable)
    - HTTP Request Defaults (server = `api.example.com`, protocol = `https`)
    - CSV Data Set Config (for dynamic payloads / item IDs)
    - Cookie Manager / Authorization Manager (if auth required)
    - Throughput Controller / Weighted Switch (to shape traffic mix)
    - Samplers:
      - HTTP Request (GET /v1/items)
      - HTTP Request (GET /v1/items/${id})
      - HTTP Request (POST /v1/items) with Body Data (from CSV)
    - Timers (think time between requests)
    - Assertions (HTTP Code = 200/201, JSON verify)
    - Backend Listener (optional - send metrics to external metrics backend)
- **Result Writer**: configure `Simple Data Writer` or set `-l` log file on CLI (CSV or JTL).

Tip: keep one "master" Thread Group for main scenario and separate Thread Groups for background traffic (e.g., scheduled batch jobs), if needed.

6.3 · Writing Realistic Workloads

- Use CSV Data Set Config to parameterize payloads and IDs so each virtual user simulates distinct users/data.
- Use timers to emulate real think time (do not hammer with constant 0ms between requests unless testing raw capacity).

- Model concurrency appropriately: ramp-up should be gradual (e.g., ramp threads over 60–300s depending on goals) to observe warm-up behavior.

6.4 · Example: Run JMeter Non-GUI and Generate HTML Dashboard

1. **Execute (non-GUI)**:

   ```
   jmeter -n -t my_test_plan.jmx -l results.jtl -e -o /tmp/jmeter-report
   ```

   - `-n`: non-GUI
   - `-t`: test plan file
   - `-l`: results log (JTL)
   - `-e -o`: generate the HTML report to specified output folder.

   This is the official recommended way to run and produce the dashboard post-run.

2. **Generate report from existing log (if you already have a .jtl):**

   ```
   jmeter -g results.jtl -o /tmp/jmeter-report
   ```

   This regenerates the HTML dashboard from a previous log file.

3. **When running distributed**: start jmeter-server on remote agents and use the master to run the same command; consult the distributed testing step-by-step guide for firewall/RMI details.

6.5 · What the HTML Dashboard Shows

The JMeter dashboard contains multiple charts and tables; important items to inspect:

- **Requests per second (throughput)** - checks handling capacity.
- **Response times (median, 90th, 95th, max)** - percentiles are more actionable than averages.
- **Errors / error types** - HTTP 4xx/5xx breakdown and error messages.
- **Active threads over time** - verify concurrency was injected as planned.
- **Apdex / satisfaction** (if configured) - mapping response time thresholds to user experience.

Use percentiles (p90/p95/p99) to characterize tail latency, averages hide spikes.

7 · Related Work

This performance testing report builds on foundational concepts and methodologies from software testing literature and practical industry best practices.