**VINH UNIVERSITY**

**SCHOOL OF ENGINEERING AND TECHNOLOGY**

****

**PROJECT REPORT ON**

**SOFTWARE TESTING AND QUALITY ASSURANCE**

**WEBSITE PERFORMANCE TESTING**

**USING GATLING**

**GROUP: 02**

|  |  |
| --- | --- |
| **Instructor**: | Cao Thanh Son, PhD. |
| **Students**: | Nguyen Tan Long, 215748020110320 |
|  | Tran Dai Viet, 215748020110356 |

**Nghe An, 04/202****5**

# ACKNOWLEDGEMENTS

I would like to express my deepest appreciation to my instructor, **Cao Thanh Son**, for his invaluable guidance, support, and encouragement throughout this project. His expertise and insightful advice have been instrumental in shaping my understanding and approach to performance testing with Gatling.

I am truly grateful for his patience, dedication, and the knowledge he has shared, which have greatly contributed to my learning and professional growth. This project would not have been possible without his guidance.

Thank you once again for your unwavering support and inspiration.

**TABLE OF CONTENTS**

[ACKNOWLEDGEMENTS 1](#_Toc195862581)

[LIST OF FIGURES 4](#_Toc195862582)

[CHAPTER 1. INTRODUCTION TO PERFORMANCE TESTING 5](#_Toc195862583)

[1.1. Overview of Performance Testing 5](#_Toc195862584)

[1.2. Benefits of Performance Testing 6](#_Toc195862585)

[1.3. Importance of Website Perfromance Testing 7](#_Toc195862586)

[1.4. Performance Testing Process 8](#_Toc195862587)

[CHAPTER 2. UNDERSTANDING GATLING 13](#_Toc195862588)

[2.1. Overview of Gatling 13](#_Toc195862589)

[2.2. Advantages and Disadvantages of Gatling 14](#_Toc195862590)

[2.3. Installing and Setting Up Gatling 15](#_Toc195862591)

[2.4. Key Components of Gatling 22](#_Toc195862592)

[2.5. Performance Testing Examples Using Gatling 25](#_Toc195862593)

[CHAPTER 3. PERFORMANCE TESTING USING GATLING 28](#_Toc195862594)

[3.1. Website 1: vinhuni.edu.vn 28](#_Toc195862595)

[3.2. Website 2: shopee.vn 31](#_Toc195862596)

[3.3. Website 3: vnexpress.net 31](#_Toc195862597)

[3.4. Website 4: tuoitre.vn 31](#_Toc195862598)

[3.5. Website 5: thegioididong.com 31](#_Toc195862599)

[3.6. Website 6: pokerogue.net 31](#_Toc195862600)

[3.7. Website 7: bienbacvn.com 31](#_Toc195862601)

[3.8. Website 8: elearning.vinhuni.edu.vn 31](#_Toc195862602)

[3.9. Website 9: wikipedia.org 31](#_Toc195862603)

[3.10. Website 10: youtube.com 32](#_Toc195862604)

[CONCLUSTIONS 33](#_Toc195862605)

[1. Achievements 33](#_Toc195862606)

[2. Limitations 33](#_Toc195862607)

[3. Future Directions 33](#_Toc195862608)

[REFERENCES 34](#_Toc195862609)

LIST OF FIGURES

[Figure 1.1 The Benefits of Website application performance testing 7](#_Toc194646289)

[Figure 1.2 Requirements analysis 8](#_Toc194646290)

[Figure 1.3 Test Planning 8](#_Toc194646291)

# CHAPTER 1. INTRODUCTION TO PERFORMANCE TESTING

## 1.1. Overview of Performance Testing

Performance testing is a method used to evaluate the speed, responsiveness, and stability of a computer, network, software application, or device under a specific workload. Organizations conduct performance tests to identify and eliminate performance-related bottlenecks, ensuring that software applications function efficiently and reliably under various conditions. By simulating real-world usage scenarios, performance testing helps teams understand system limitations and optimize performance before deployment.

The primary goal of performance testing is to detect and resolve bottlenecks in software applications, contributing to overall software quality. Without proper performance testing, systems may suffer from slow response times, increased downtime, and inconsistent experiences across different users and operating systems (OS). These issues can negatively impact business operations, leading to customer dissatisfaction and revenue loss. By verifying that a system meets speed, responsiveness, and stability requirements under various workloads, performance testing helps enhance UX and improve overall system efficiency.

Performance testing should be conducted after functional testing is completed to ensure that the core features of the application work as intended before assessing performance. Developers can write performance tests as part of the code review process, and these test scenarios can be transferred between different environments - such as from development teams testing in a live environment to operations teams monitoring production systems. Performance testing can be carried out in both controlled lab conditions and real-world production environments, allowing for a comprehensive assessment of system performance.

During performance testing, key system requirements should be identified and evaluated to ensure smooth operation under different levels of stress. Common parameters include processing speed, data transfer rates, network bandwidth, throughput, workload efficiency, and system reliability. Additionally, performance testing helps organizations assess the impact of infrastructure changes, software updates, and traffic spikes, ensuring that systems remain stable and performant over time.

For example, an organization may measure the response time of a software application when a user initiates an action, such as submitting a form or processing a transaction. The same test can be performed on a larger scale to assess system performance under high user loads and unexpected surges in demand. If response times are slow or erratic, developers must analyze the system to pinpoint and resolve the bottleneck, ensuring a seamless user experience.

## 1.2. Benefits of Performance Testing

Organizations implement performance testing for various reasons, including the following:

* **Identifying bottlenecks** – Performance testing helps diagnose and allocate computing or communication bottlenecks within a system. Bottlenecks are single points or components that hinder overall performance. For instance, even the most powerful computer may struggle with web performance if the available bandwidth is too low. Slow data transfer rates can be caused by hardware limitations or software-related issues, such as excessive background applications or corrupted files in a web browser.
* **Detecting software performance issues** – This type of testing helps pinpoint the nature and location of performance-related problems in software applications. It highlights potential failure points or areas where an application may lag. Additionally, organizations can use performance testing to prepare for predictable high-traffic events and ensure system stability.
* **Validating vendor claims** – Performance testing verifies whether a system meets the specifications claimed by its manufacturer or vendor. It also allows organizations to compare multiple devices or software solutions to determine which performs better under specific conditions.
* **Providing insights for stakeholders** – The results of performance testing keep project stakeholders informed about critical aspects of application performance, including speed, stability, and scalability. This information helps guide decision-making and optimizations.
* **Preventing reputation damage** – Releasing an application without proper performance testing can result in poor system performance, leading to negative user experiences and bad word-of-mouth. Thorough testing helps avoid these risks and ensures a more reliable product.
* **Comparing systems** – Organizations can use performance testing to compare multiple software systems in terms of speed, responsiveness, and stability, helping them choose the best solution for their needs.

## 1.3. Importance of Website Perfromance Testing

Website performance testing is crucial for ensuring a smooth and seamless user experience. It helps identify bottlenecks, optimize response times, and improve overall system reliability. Some key reasons why website performance testing is essential include.

**User experience**: Slow-loading websites lead to poor user experience, reducing engagement and increasing bounce rates. A fast and responsive website improves user retention and satisfaction.

**SEO ranking**: Search engines prioritize fast-loading websites, directly affecting visibility and ranking in search results. Google’s algorithms favor sites with better performance, leading to higher organic traffic.

**Scalability**: Performance testing ensures that a website can handle a growing number of users and concurrent requests without degradation. It helps businesses prepare for peak traffic events, such as sales, product launches, or seasonal demand spikes.

**Revenue impact**: For e-commerce and business websites, even a slight delay in loading time can result in revenue loss. Studies show that a one-second delay in page load time can lead to significant drops in conversion rates.

**System stability**: Identifying performance bottlenecks early helps prevent crashes and downtime during peak traffic periods. Load and stress testing can reveal weaknesses in infrastructure and code that could lead to service disruptions.

**Compliance and SLAs**: Many businesses operate under service-level agreements (SLAs) that mandate a certain level of website performance. Failing to meet these requirements could lead to legal issues, financial penalties, and loss of customer trust.

**A diagram of a web application performance

AI-generated content may be incorrect.**

Figure 1.1. The Benefits of Website application performance testing

By conducting regular performance testing, organizations can maintain high-performing websites, enhance customer satisfaction, and achieve business goals effectively. Performance testing also plays a critical role in proactive maintenance, allowing teams to fix issues before they impact end users.

## 1.4. Performance Testing Process

Performance testing involves a systematic approach to evaluating a website’s responsiveness, stability, scalability, and resource usage under varying loads. The key steps in the performance testing process include:

**Step1: Requirement analysis**

* Define the objectives of performance testing (e.g., response time, load handling capacity, scalability requirements).
* Identify key performance indicators (KPIs), such as throughput, latency, error rates, and resource utilization.
* Establish baseline performance metrics to compare test results.

**A screen shot of a game

AI-generated content may be incorrect.**

Figure 1.2. Requirements analysis

**Step 2: Test planning**

* Choose appropriate performance testing tools (e.g., Gatling, JMeter, LoadRunner) based on application type and infrastructure.
* Determine test scenarios based on expected user behavior, peak loads, and potential failure points.
* Set up test environments to mimic production settings, including network conditions, database loads, and hardware configurations.
* Define different types of tests, such as load testing, stress testing, spike testing, endurance testing, and scalability testing.

Figure 1.3. Test Planning

**Step 3: Scripting and test case development**

* Develop test scripts using tools like Gatling to simulate realistic user interactions and workflows.
* Define load profiles, including ramp-up time, concurrent users, and peak traffic conditions.
* Include different user personas and behaviors to ensure comprehensive coverage.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 1.4. Test Planning

**Step 4: Test execution**

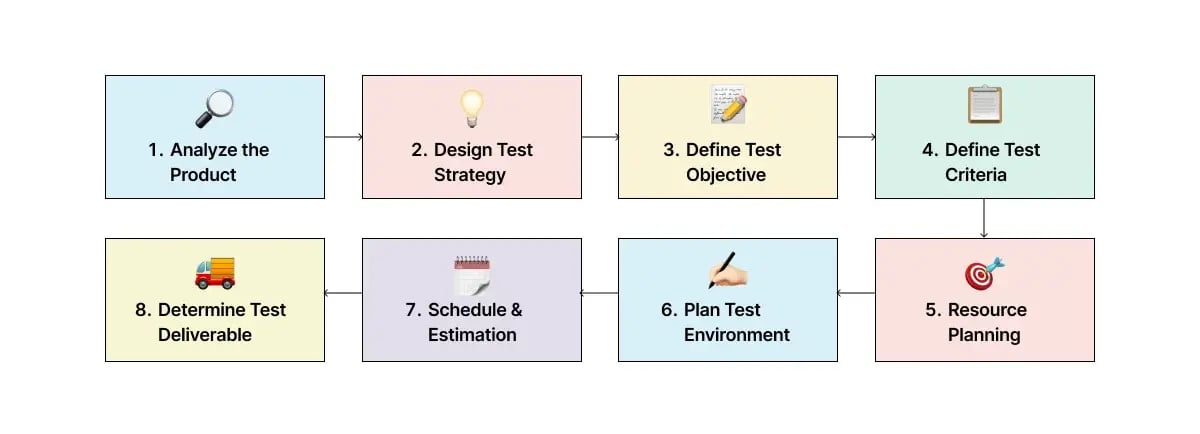
* Run test scripts and gradually increase load to observe system behavior under normal and extreme conditions.
* Monitor system performance using logging, real-time monitoring tools, and built-in performance analytics.
* Capture key metrics such as response time, error rates, CPU and memory usage, and database query performance. 

Figure 1.5. Test Planning

**Step 5: Performance analysis and bottleneck identification**

* Analyze test results to detect performance bottlenecks.
* Identify slow database queries, inefficient code, server resource limitations, and network latency issues.
* Compare test results with baseline metrics to measure improvements or degradations.
* Use profiling tools to pinpoint inefficient code and optimize system performance.

**Step 6: Optimization and Retesting**

* Implement necessary optimizations, such as caching, load balancing, database indexing, and code improvements.
* Optimize frontend performance by minimizing HTTP requests, compressing images, and leveraging content delivery networks (CDNs).
* Retest after optimizations to validate improvements and ensure performance goals are met.
* Conduct iterative testing cycles to fine-tune system performance.

**Step 7: Reporting and recommendations**

* Generate detailed test reports with insights, visual representations, and recommendations for stakeholders.
* Provide actionable steps to address identified performance issues and potential risks.
* Share findings with development, operations, and business teams for informed decision-making and future improvements.
* Document lessons learned and best practices for continuous performance optimization.

Following this structured approach ensures that performance testing is effective, allowing websites to handle real-world traffic efficiently while maintaining optimal user experience. It also helps organizations build robust systems that can scale seamlessly and remain resilient under high demand.

# CHAPTER 2. UNDERSTANDING GATLING

## 2.1. Overview of Gatling

Gatling is an open-source performance testing tool designed to evaluate the load-handling capabilities of web applications and APIs. Developed in **Scala**, it leverages an **asynchronous programming model** and an **event-driven architecture**, making it more efficient than traditional performance testing tools such as JMeter.



Figure 2.1. Gatling Logo

Gatling enables organizations to simulate real-world user interactions, assess system performance under varying loads, and identify performance bottlenecks. The tool provides **comprehensive performance reports**, which help developers and testers optimize system stability and responsiveness. Due to its lightweight nature and **seamless integration with CI/CD pipelines**, Gatling is widely used in DevOps environments.

### 2.1.1. Function analysis

Gatling offers several key functionalities that enhance its effectiveness as a performance testing tool:

**Load Simulation:** Gatling can simulate thousands of concurrent users interacting with a system to evaluate its performance under stress.

**Real-Time Reporting:** The tool generates **detailed graphical reports**, enabling teams to analyze system behavior and detect inefficiencies.

**Event-Driven Model:** Unlike thread-based tools, Gatling’s non-blocking architecture ensures minimal resource consumption while handling large workloads.

**Scenario Scripting:** Gatling uses a **Scala-based domain-specific language (DSL)** to define test scenarios in a structured and readable format.

**CI/CD Integration:** The tool is compatible with **Jenkins, GitLab CI, and other DevOps tools**, allowing organizations to automate performance testing.

**Protocol Support:** Gatling primarily supports **HTTP-based testing** but can be extended to other protocols such as WebSockets and JMS.

### 2.1.2. Design analysis

Gatling’s architecture is designed to maximize efficiency and scalability, consisting of three main components:

**Simulation engine:** The core processing unit responsible for executing test scenarios and managing virtual users asynchronously.

**Recorder:** A built-in tool that captures user interactions and converts them into reusable test scripts.

**Results analyzer:** A reporting module that generates **HTML-based visual reports**, displaying response times, failure rates, and system performance metrics.

The adoption of an **asynchronous event-driven model** enables Gatling to efficiently handle a high number of concurrent users without excessive consumption of CPU or memory resources. This design choice makes it particularly suitable for modern **high-traffic web applications**.

## 2.2. Advantages and Disadvantages of Gatling

Gatling is a powerful performance testing tool designed for developers and testers to simulate high-load scenarios on web applications. While it offers numerous advantages, it also comes with certain limitations. Below is a detailed analysis of its benefits and drawbacks.

|  |  |
| --- | --- |
| **Advantages** | **Disadvantages** |
| **High Performance and Scalability**: Uses an asynchronous, event-driven architecture, allowing it to handle high loads with minimal system resource consumption. | **Learning Curve**: Requires knowledge of Scala, which can be challenging for testers unfamiliar with programming. |
| **Developer-Friendly and Code-Based Approach**: Provides a Domain-Specific Language (DSL) in Scala, making scripts readable and reusable. | **Limited Protocol Support**: Supports HTTP, WebSockets, and JMS but lacks built-in support for FTP, SMTP, and SOAP. |
| **Rich Reporting and Visualization**: Automatically generates interactive HTML reports with key performance metrics. | **Higher Setup Complexity**: Requires manual installation and configuration, making it less user-friendly for beginners. |
| **CI/CD Integration**: Works seamlessly with Jenkins, GitLab CI, and Bamboo, enabling automated performance testing. | **Limited GUI Support**: Lacks a graphical interface, which can be a disadvantage for non-technical testers. |
| **Support for HTTP and WebSocket Protocols**: Can test various types of applications, including RESTful APIs and real-time services. | **Limited Community Resources Compared to JMeter**: Smaller community means fewer tutorials and third-party integrations. |
| **Have a Web Version: Gatling has a website for users to run tests on without coding knowledges.** | **Paid access only**: You only get one free token to run on the web version. |
| **Open-Source and Extensible**: Strong community backing with various plugins and extensions for customization. |  |

Despite these limitations, Gatling remains a preferred choice for performance testing in agile development environments, especially where automation and integration with CI/CD pipelines are essential.

## 2.3. Installing and Setting Up Gatling

Gatling is a powerful performance testing tool that requires proper installation and setup to be used effectively. This section provides a step-by-step guide to installing Gatling, setting up the necessary dependencies, and configuring the tool for performance testing.

### 2.3.1. Prerequisites

Before installing Gatling, ensure the following requirements are met:

* **Java Development Kit (JDK 8 or higher)** installed and configured.
* **Sufficient system resources** (CPU, RAM) to execute high-load simulations.
* **Scala (optional):** While not required, familiarity with Scala helps in writing advanced test scripts.

### 2.3.2. Installation Steps

Gatling is available in two primary editions:

* **Gatling Open-Source Edition** (free to use)
* **Gatling Enterprise Edition** (paid version with additional features)

For this document, we will focus on installing the **open-source version**.

**Step 1: Download Gatling**

* Visit the official Gatling website: <https://gatling.io/>
* Navigate to the **Download** section and select the latest version of Gatling.

A screenshot of a computer

AI-generated content may be incorrect.

* Choose the appropriate distribution based on the operating system:

**+ Windows:** ZIP archive

**+ macOS/Linux:** TAR.GZ archive

A screenshot of a computer

AI-generated content may be incorrect.

**Step 2: Extract the Downloaded File**

* On **Windows**, use a file extraction tool (such as WinRAR or 7-Zip) to extract the ZIP archive to a preferred directory (e.g., C:\Gatling).
* On **macOS/Linux**, open a terminal and use the following command to extract the TAR.GZ archive: **tar -xvzf gatling-<version>.tar.gz -C /opt/**

This command extracts Gatling into the /opt/ directory, which is commonly used for software installations on Linux-based systems.

**Step 3: Verify Installation**

* Navigate to the Gatling bin directory: cd gatling-<version>/bin
* Run the following command to verify Gatling is installed correctly: ./gatling.sh

+ If successful, the Gatling interactive menu will appear in the terminal.

+ On **Windows**, use gatling.bat instead: gatling.bat

### 2.3.3. Setting Up Gatling Recorder

Gatling provides a **built-in recorder** that captures user interactions and converts them into test scripts. This feature is useful for testers who want to automate performance tests without writing scripts manually.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 2.2. Gatling Recorder UI

**Launching the Recorder**

* **On Linux/macOS:** ./recorder.sh
* **On Windows:** recorder.bat
* The Gatling Recorder interface will open, allowing users to configure their test recording preferences.

**Configuring the Recorder**

**Step 1: Select the Recorder Mode**

* + **HTTP Proxy Mode:** Captures web traffic through a proxy (useful for browser-based interactions).
  + **HAR File Converter Mode:** Converts **HAR files** (HTTP Archive files) from browsers into Gatling test scripts.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 2.3. Gatling Recorder when running

**Step 2: Set the Output Directory**: Define where the recorded test script will be stored. The default location is user-files/simulations/.

**Step 3: Define the Simulation Name**: Enter a meaningful name for the recorded test scenario (e.g., LoginSimulation).

**Step 4: Start Recording**

* + Click **Start** to begin capturing interactions.
  + Perform the required actions on the website or application.
  + Click **Stop** once the interaction is complete.

**Step 5: Generate the Test Script**

* + After stopping the recording, Gatling will generate a test script in Scala format.
  + The script will be stored in the **simulations** directory and can be modified for further customization.

### 2.3.4. Configuring Gatling for Automated Testing

For efficient test execution, Gatling can be configured within **CI/CD pipelines** or executed in batch mode.

**Running a Gatling Test Manually:**

Step 1: Navigate to the bin/ directory: cd gatling-<version>/bin

Step 2: Execute a test script: ./gatling.sh -s <SimulationClassName>

Example: ./gatling.sh -s LoginSimulation

Step 3: The test results will be displayed in the terminal, and a detailed **HTML report** will be generated in the results/ directory.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 2.4. HTML Report

**Integrating Gatling with CI/CD Pipelines:** Gatling can be integrated with **Jenkins, GitLab CI/CD, GitHub Actions, and other DevOps tools** to automate performance testing.

**Jenkins Integration Example:**

* + Install the **Gatling Plugin** from Jenkins Plugin Manager.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 2.5 Install Gatling Plugin in Jenkins

* + Create a new **Jenkins Pipeline** and define the Gatling execution step in the Jenkinsfile:

pipeline {

agent any

stages {

stage('Performance Test') {

steps {

sh './gatling.sh -s LoginSimulation'

}

}

}

}

* + This setup will trigger **automated performance tests** whenever a new code deployment occurs.

### 2.3.5. Troubleshooting Common Issues

| **Issue** | **Possible Cause** | **Solution** |
| --- | --- | --- |
| **"Command not found" when running Gatling** | The execution path is incorrect | Navigate to the correct bin/ directory |
| **Java version issues** | JDK is not installed or incorrectly configured | Install JDK 8+ and set JAVA\_HOME environment variable |
| **Tests fail with "Connection Refused"** | The target application is not running | Verify that the application URL is correct and accessible |
| **Gatling reports not generating** | Insufficient system resources or incorrect configuration | Increase available memory and check log files for errors |

## 2.4. Key Components of Gatling

To effectively use Gatling for performance testing, it is crucial to understand its core components. These components define how tests are structured, executed, and analyzed.

Below are **Gatling** key components:

### 2.4.1. Simulation

A **simulation** is the primary script that defines a performance test in Gatling. Written in Scala, it specifies the test scenario, including user behavior, load patterns, and assertions. Each simulation consists of multiple elements, such as HTTP requests, pauses, loops, and feeders.

### A screen shot of a computer code AI-generated content may be incorrect.

Figure 2.6. An example of Simulation Script

### 2.4.2. Scenario

A **scenario** represents the sequence of actions that virtual users will perform during the test. Scenarios allow for realistic user behavior modeling, including navigation through different pages, performing transactions, and logging in/out. Complex scenarios can be defined using loops, conditionals, and feeder data to simulate diverse user behaviors.

### 2.4.3. HTTP Protocol Configurations

This configuration defines how HTTP requests are sent and processed. It includes setting the base URL, defining headers, setting up authentication methods, and handling cookies. It ensures that tests interact with the system under realistic conditions.

A screen shot of a computer code

AI-generated content may be incorrect.

Figure 2.7 Example of HTTP Protocol Config

### 2.4.4. Feeders

**Feeders** supply dynamic test data, enhancing the realism of performance tests. They support various data formats, including CSV, JSON, and JDBC. Feeders enable different users to have unique input values, avoiding repetitive test scenarios that might not reflect real-world usage.



Figure 2.8. Example of Feeders

### 2.4.5. Injectors (Load Injection Profiles)

**Injectors** determine how users are introduced into the system during the test. Gatling provides multiple strategies, such as:

* **atOnceUsers(n)**: Instantly launches *n* users.
* **rampUsers(n) during (t seconds)**: Gradually increases users over time.
* **constantUsersPerSec(n) during (t seconds)**: Maintains a steady user load.
* **heavisideUsers(n) during (t seconds)**: Gradually ramps up users following a heaviside function.
* **incrementUsersPerSec(5).times(5).eachLevelLasting(10.seconds)**: Gradually increases user count over a series of stages.

### 2.4.6 Checks and Assertions

Checks are used to validate responses from the server during the test, ensuring that requests return the expected results. Assertions define success criteria for the test based on performance metrics.



Figure 2.9. Example of Checks

Assertions allow for automated validation of test results:

A close-up of a computer screen

AI-generated content may be incorrect.

Figure 2.10. Example of Assertions

### 2.4.7 Reports and Logs

Gatling generates detailed HTML reports containing key performance metrics, such as response time distributions and error rates. Logs allow deeper analysis of test results. The reports includes **response time distribution**, **percentiles**, **active users over time**, **requests per seconds**, **failure rates** and more.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 2.11. Gatling HTML Report

### 2.4.8 Gatling Recorder

The **Gatling Recorder** is a GUI tool that captures browser interactions and converts them into test scripts. It simplifies script creation, especially for those unfamiliar with Scala. It allows users to record HTTP traffic via a proxy and automatically generates a Gatling test script based on the recorded session.

### 2.4.9 Simulation Execution and Result Analysis

After writing a test script, execute it using the Gatling command:

**./gatling.sh -s BasicSimulation**

Once the test completes, Gatling generates an interactive report with visual graphs showing test results.

A graph with lines and dots

AI-generated content may be incorrect.

Figure 2.12. Gatling Visual Graph

## 2.5. Performance Testing Examples Using Gatling

The following example demonstrates a **basic Gatling script** for testing the homepage response time of a web application.

import io.gatling.core.scenario.Simulation

import io.gatling.core.Predef.\_

import io.gatling.http.Predef.\_

class BasicSimulation extends Simulation {

val httpProtocol = http

.baseUrl("https://example.com") // Target application URL

.header("Accept", "application/json")

val scn = scenario("Homepage Load Test")

.exec(

http("Homepage Request")

.get("/")

.check(status.is(200)) // Verify HTTP response status

)

setUp(

scn.inject(atOnceUsers(100)) // Simulate 100 concurrent users

).protocols(httpProtocol)

}

**Explanation:**

* Defines a **performance testing scenario** in Scala.
* Sets the **base URL** for testing.
* Simulates **100 concurrent users** accessing the homepage.
* Checks if the response status is **HTTP 200 (OK)**.

A screenshot of a computer

AI-generated content may be incorrect.

**Running the Test**

To execute the test, navigate to the bin/ directory and run: ./gatling.sh -s BasicSimulation

Upon completion, Gatling will generate a **detailed HTML report** displaying response times, error rates, and system performance metrics.

A screenshot of a computer

AI-generated content may be incorrect.

# CHAPTER 3. PERFORMANCE TESTING USING GATLING

## 3.1. Website 1: vinhuni.edu.vn

### 3.1.1. Developing an Execution Plan

**Identifying test objectives**

The main objective of performance testing for *vinhuni.edu.vn* is to measure how well the website performs under different user loads. The focus is on evaluating its responsiveness, stability, and resource consumption during common user activities such as accessing the homepage, searching for academic information, logging in, and downloading files.

**Defining the test scope**

The scope of testing includes:

* High-traffic user actions like homepage access, news viewing, academic search.
* Backend services: login, file downloads, content rendering.
* Features that are essential for students, faculty, and guests.
* Excludes non-critical features like user profile customization.

The following key functions are included in the scope:

* Homepage loading
* News section browsing
* Academic program search
* User login (students, teachers)
* Viewing announcement details
* PDF document downloads
* Navigation between multiple pages
* Simulating extended sessions
* Combined flows (e.g., search then download)
* Handling peak loads or sudden spikes

Test conditions are applied under constraints such as server capacity, testing time, and a staging environment that mimics production.

### 3.1.2. Designing Performance Test Cases

All test cases are designed to run with **10 different levels of virtual users**, from 100 to 1000 (step by 100). Each test case will therefore be executed 10 times, resulting in a total of **100 test executions** for this website.

**Test Case Summary:**

| **Test Case ID** | **Test Name** | **Scenario Description** | **Expected Outcome** |
| --- | --- | --- | --- |
| TC\_VU\_001 | Homepage Load Test | Users access the homepage concurrently | Avg. response < 2s, no errors |
| TC\_VU\_002 | News Section Access Test | Users browse the latest news | 95% requests under SLA, stable response |
| TC\_VU\_003 | Academic Search Test | Users search for training/academic programs | Result within 3s, no errors |
| TC\_VU\_004 | Login Test | Login using valid accounts | >98% success, login < 2s |
| TC\_VU\_005 | Announcement Detail View Test | Open and read announcement details | 95% successful requests |
| TC\_VU\_006 | PDF Download Test | Simultaneous document downloads | >95% download success, no timeout |
| TC\_VU\_007 | Multi-Page Navigation Test | Homepage → Search → Detail view | Flow completed in under 6s |
| TC\_VU\_008 | Long Session Simulation | 10-min session with repeated navigation | No memory leaks, stable CPU/memory |
| TC\_VU\_009 | Search + Download Combo Test | Users search and download files | Smooth operation, no crashes |
| TC\_VU\_010 | Peak Load Stress Test | Ramp-up from 100 to 1000 users in 1 minute | System remains responsive, no crashes |

Each test case is configured to simulate:

* 100 users
* 200 users
* 300 users

...

* up to 1000 users

That makes 10 load levels x 10 test cases = **100 performance test runs**.

### 3.1.3. Executing Performance Tests

Performance tests are executed using Gatling. Each test case is scripted using Scala DSL and configured with different user loads (from 100 to 1000). The tests are launched via Maven in a controlled test environment that mirrors the production setup of vinhuni.edu.vn.

**Each test run includes:**

* Ramp-up time: gradually introducing users over 30 seconds to 1 minute depending on the scenario.
* Execution duration: typically 3–10 minutes per run depending on flow complexity.
* Data input: user credentials, search keywords, download file paths—provided through CSV feeders.
* Monitoring: Server resource usage is tracked using tools like Grafana and Prometheus during each test.

Each test run is logged, and results are stored for later analysis.

### 3.1.4. Analyzing Test Results

### 3.1.5. Final Evaluation of Test Results

## 3.2. Website 2: shopee.vn

### 3.2.1. Developing an Execution Plan

### 3.2.2. Designing Performance Test Cases

### 3.2.3. Executing Performance Tests

### 3.2.4. Analyzing Test Results

### 3.2.5. Final Evaluation of Test Results

## 3.3. Website 3: vnexpress.net

### 3.3.1. Developing an Execution Plan

### 3.3.2. Designing Performance Test Cases

### 3.3.3. Executing Performance Tests

### 3.3.4. Analyzing Test Results

### 3.3.5. Final Evaluation of Test Results

## 3.4. Website 4: tuoitre.vn

### 3.4.1. Developing an Execution Plan

### 3.4.2. Designing Performance Test Cases

### 3.4.3. Executing Performance Tests

### 3.4.4. Analyzing Test Results

### 3.4.5. Final Evaluation of Test Results

## 3.5. Website 5: thegioididong.com

### 3.5.1. Developing an Execution Plan

Identifying test objectives

The main goal of this performance test plan is to assess the responsiveness, stability, and scalability of thegioididong.com, a popular Vietnamese electronics e-commerce platform. The tests will simulate real-world user behaviors such as searching for products, navigating through categories, viewing product details, and simulating shopping cart actions. The results will help identify slow-loading components, bottlenecks during high traffic, and assess the website's ability to scale under load.

**Defining the test scope**

The scope of testing includes:

- Homepage load performance.

- Product search functionality.

- Category page navigation.

- Product detail viewing.

- Shopping cart and comparison actions.

- Handling traffic spikes and prolonged use.

Test conditions are applied under constraints such as server capacity, testing time, and a staging environment that mimics production.

### 3.5.2. Designing Performance Test Cases

All test cases are designed to run with **5 different levels of virtual users**, from 100 to 1000 (step by 100). Each test case will therefore be executed 10 times, resulting in a total of **50 test executions** for this website.

**Test Case Summary:**

| **Test Case ID** | **Test Name** | **Scenario Description** | **Expected Outcome** |
| --- | --- | --- | --- |
| TC\_TGDD\_001 | Homepage Load Test | Users access the homepage concurrently | Avg. response < 2s, no errors |
| TC\_TGDD\_002 | Product Search Test | Users search for a keyword (e.g., iPhone) | Search results < 2s |
| TC\_TGDD\_003 | Category Browse Test | Navigate to phones category | Page loads < 2s |
| TC\_TGDD\_004 | Pagination Test | Browse page 2 of category list | Next page loads < 2s |
| TC\_TGDD\_005 | Product Detail View | View details of a product | Product page loads < 2s |
| TC\_TGDD\_006 | Promotion Page Test | Access the promotions page | Promo page loads < 2s |
| TC\_TGDD\_007 | Store Locator Test | View the store locator system | Locator loads < 2s |
| TC\_TGDD\_008 | Compare Products Test | Compare two selected phones | Comparison loads < 2s |
| TC\_TGDD\_009 | Image Load Test | Load product images or banners | Images load < 1.5s |
| TC\_TGDD\_010 | Cart Simulation Test | Add product to cart and open cart | Cart interaction < 3s |

Each test case is configured to simulate:

* 100 users
* 500 users
* 1000 users
* 2000 users
* 5000 users

That makes 5 load levels x 10 test cases = **50 performance test runs**

### 3.5.3. Executing Performance Tests

### 3.5.4. Analyzing Test Results

### 3.5.5. Final Evaluation of Test Results

## 3.6. Website 6: pokerogue.net

### 3.6.1. Developing an Execution Plan

**Identifying test objectives**

The main goal of this performance test plan is to assess the responsiveness, stability, and scalability of **pokerogue.net**, a browser-based roguelike Pokémon game. The tests will simulate real-world scenarios such as peak concurrent gameplay, long user sessions, and cross-device compatibility to ensure the game performs optimally under stress. The results will help identify performance bottlenecks, potential memory leaks, and load-handling capabilities.

**Defining the test scope**

The scope of testing includes:

- Page load performance

- Game initialization and character selection

- Battle system responsiveness

- Inventory management operations

- System behavior under prolonged use and traffic surges

- Handling peak load or system spike

Test conditions are applied under constraints such as server capacity, testing time, and a staging environment that mimics production.

### 3.6.2. Designing Performance Test Cases

All test cases are designed to run with **5 different levels of virtual users**, from 100 to 1000 (step by 100). Each test case will therefore be executed 10 times, resulting in a total of **50 test executions** for this website.

**Test Case Summary:**

| **Test Case ID** | **Test Name** | **Scenario Description** | **Expected Outcome** |
| --- | --- | --- | --- |
| TC\_PR\_001 | Homepage Load Test | Users access the homepage concurrently | Avg. response < 2s, no errors |
| TC\_PR\_002 | Login Test | Login using valid accounts | >98% success, login < 2s |
| TC\_PR\_003 | Menu Navigate Test | Navigate through different menus | Avg. response < 2s, no errors |
| TC\_PR\_004 | Game Start Test | User start a new save from homepage | >98% success, avg response <3s, no errors |
| TC\_PR\_005 | Pokemon Select Test | User choosing starter pokemons | Selection complete in <2s, no errors |
| TC\_PR\_006 | Pokemon Filter Test | User filter pokemons for choosing | Filtered list loaded in <2s, no errors |
| TC\_PR\_007 | Battle System Test | User go through a normal battle | Battle actions load in <2s, no errors |
| TC\_PR\_008 | Inventory Test | User view/moving inventory items | Inventory load in <2s, moving items in <2s |
| TC\_PR\_009 | Save system test | User saving their games | Saved in <2s, no errors |
| TC\_PR\_010 | Gacha system test | User gacha for pokemon eggs | Gacha results loaded in <5s, no errors. |

Each test case is configured to simulate:

* 100 users
* 500 users
* 1000 users
* 2000 users
* 5000 users

That makes 5 load levels x 10 test cases = **50 performance test runs**

### 3.6.3. Executing Performance Tests

#### 3.6.3.1 Homepage Load Test

| **User Load** | **Min Response Time (ms)** | **Max Response Time (ms)** | **Avg Response Time (ms)** | **Avg Success Rate (%)** | **Avg Error Rate (%)** | **Avg Throughput (req/s)** |
| --- | --- | --- | --- | --- | --- | --- |
| **12AM** | | | | | | |
| 100 | 460 | 680 | 586 | 100 | 0 | 100 |
| 500 | 472 | 1438 | 974 | 100 | 0 | 250 |
| 1000 | 1060 | 7543 | 3215 | 97.5 | 2.5 | 197.35 |
| 2000 | 1582 | 10218 | 5732 | 78.8 | 21.2 | 276.71 |
| 5000 | 1922 | 27810 | 9685 | 70.17 | 29.83 | 284.52 |
| 6AM | | | | | | |
| 100 | 574 | 835 | 717 | 100 | 0 | 100 |
| 500 | 561 | 1722 | 1259 | 100 | 0 | 250 |
| 1000 | 1308 | 9047 | 4031 | 94.2 | 5.8 | 193.42 |
| 2000 | 2042 | 12887 | 6943 | 74.5 | 25.5 | 278.63 |
| 5000 | 2418 | 33892 | 11382 | 66.17 | 33.83 | 287.77 |
| **10AM** | | | | | | |

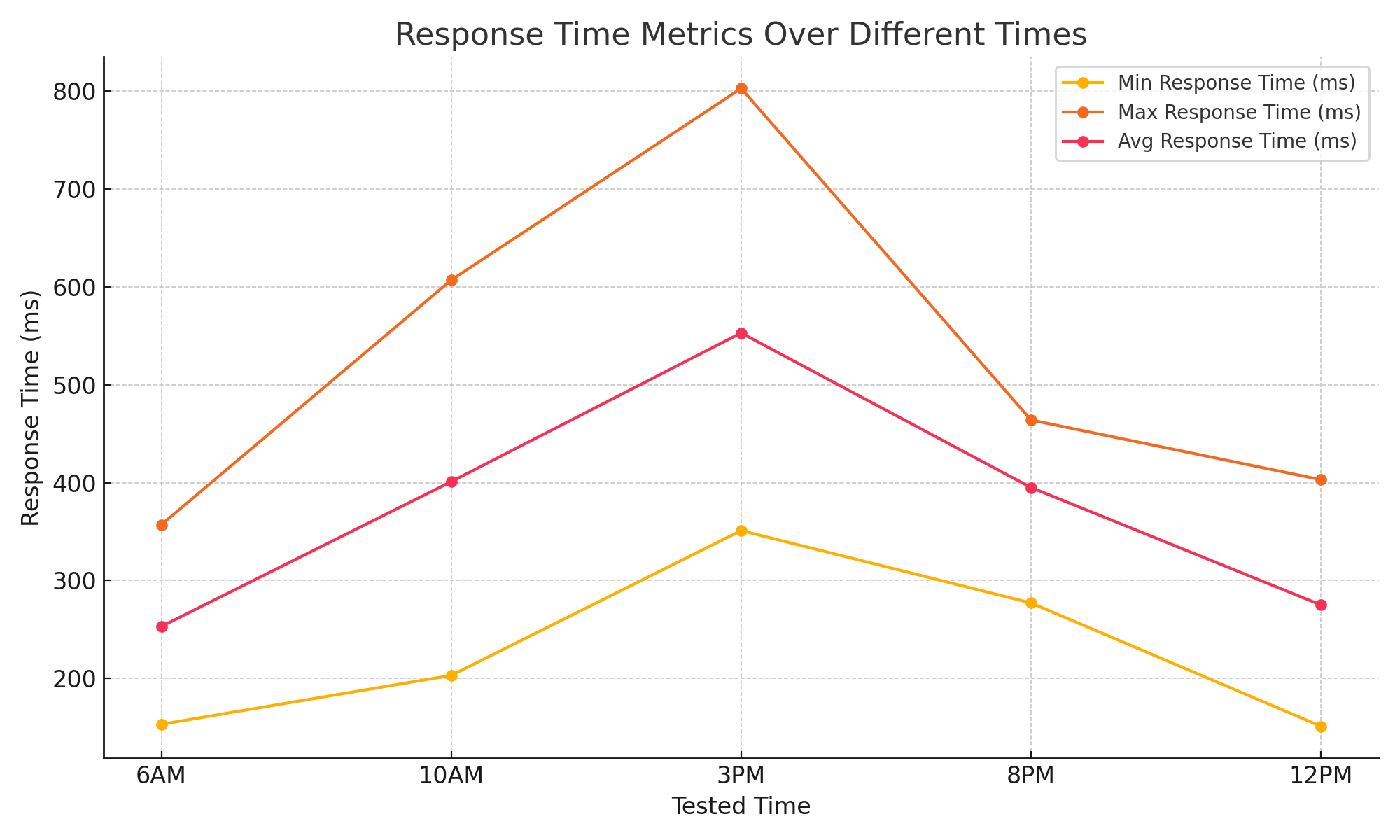
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 100 | 583 | | 841 | | 719 | | 100 | | 0 | | 100 | |
| 500 | 557 | | 1753 | | 1272 | | 100 | | 0 | | 250 | |
| 1000 | 1314 | | 9120 | | 4087 | | 93.79 | | 6.21 | | 189.69 | |
| 2000 | 2048 | | 13027 | | 7018 | | 73.82 | | 26.18 | | 274.55 | |
| 5000 | 2467 | | 34076 | | 11513 | | 65.73 | | 34.27 | | 284.3 | |
| **3PM** | | | | | | | | | | | | |
| 100 | 555 | | 809 | | 703 | | 100 | | 0 | | 100 | |
| 500 | 541 | | 1695 | | 1210 | | 100 | | 0 | | 250 | |
| 1000 | 1268 | | 8878 | | 3894 | | 94.9 | | 5.1 | | 200 | |
| 2000 | 1966 | | 12596 | | 6839 | | 75.4 | | 24.6 | | 285.71 | |
| 5000 | 2343 | | 32835 | | 11160 | | 66.85 | | 33.15 | | 294.12 | |
| **8PM** | | | | | | | | | | | | |
| 100 | | 472 | | 691 | | 603 | | 100 | | 0 | | 100 |
| 500 | | 461 | | 1448 | | 996 | | 100 | | 0 | | 250 |
| 1000 | | 1105 | | 7614 | | 3315 | | 96.91 | | 3.09 | | 219.34 |
| 2000 | | 1662 | | 10285 | | 5631 | | 81.38 | | 18.62 | | 316.53 |
| 5000 | | 1961 | | 26897 | | 9292 | | 75.66 | | 24.44 | | 324.59 |

A graph with a line

AI-generated content may be incorrect.

#### 3.6.3.2 Login Test

| **Tested Time** | **Min Response Time (ms)** | **Max Response Time (ms)** | **Avg Response Time (ms)** | **Total Request** | **Avg Success Rate (%)** | **Avg Error Rate (%)** | **Avg Throughput (req/s)** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **6AM** | 153 | 357 | 253 | 9 | 66% | 33% | 4.5 |
| **10AM** | 203 | 607 | 401 | 9 | 66% | 33% | 4.5 |
| **3PM** | 351 | 803 | 553 | 9 | 66% | 33% | 4.5 |
| **8PM** | 277 | 464 | 395 | 9 | 66% | 33% | 4.5 |
| **12AM** | 151 | 403 | 275 | 9 | 66% | 33% | 4.5 |



#### 3.6.3.3 Menu Navigate Test

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **User Load** | | **Min Response Time (ms)** | | **Max Response Time (ms)** | | **Avg Response Time (ms)** | | **Avg Success Rate (%)** | | **Avg Error Rate (%)** | | **Avg Throughput (req/s)** | |
| **Time Slot: 6AM** | | | | | | | | | | | | | |
| 100 | | 20.8 | | 912.4 | | 127.3 | | 99.63 | | 0 | | 94.2 | |
| 500 | | 31.2 | | 3427.5 | | 412.7 | | 97.24 | | 0 | | 203.8 | |
| 1000 | | 45.6 | | 13207.9 | | 532.1 | | 85.72 | | 0 | | 276.3 | |
| 2000 | | 71.9 | | 24883.2 | | 912.8 | | 73.64 | | 0 | | 342.7 | |
| 5000 | | 112.4 | | 39003.6 | | 2287.5 | | 58.93 | | 0 | | 396.4 | |
| **Time Slot: 10AM** | | | | | | | | | | | | | |
| 100 | | 23 | | 1347 | | 162 | | 98.7 | | 0 | | 53.8 | |
| 500 | | 48 | | 6215 | | 483 | | 94.2 | | 0 | | 127.4 | |
| 1000 | | 81 | | 33892 | | 1315 | | 56.4 | | 3.09 | | 387.5 | |
| 2000 | | 137 | | 47852 | | 2945 | | 39.8 | | 18.62 | | 512.6 | |
| 5000 | | 218 | | 62409 | | 5873 | | 28.3 | | 24.44 | | 603.1 | |
| **Time Slot: 3PM** | | | | | | | | | | | | | |
| 100 | | 23.1 | | 1427.5 | | 147.9 | | 98.47 | |  | | 88.3 | |
| 500 | | 37.6 | | 6112.8 | | 453.2 | | 94.82 | |  | | 182.7 | |
| 1000 | | 52.9 | | 17449.1 | | 587.4 | | 79.23 | |  | | 256.4 | |
| 2000 | | 83.5 | | 29276.3 | | 1287.6 | | 67.15 | |  | | 327.9 | |
| 5000 | | 132.7 | | 43856.4 | | 2763.2 | | 52.41 | |  | | 378.2 | |
| **Time Slot: 8PM** | | | | | | | | | | | | |
| 100 | 28.4 | | 2115.7 | | 206.3 | | 97.12 | |  | | 76.5 | |
| 500 | 44.9 | | 8876.2 | | 587.4 | | 89.13 | |  | | 142.6 | |
| 1000 | 63.7 | | 23997.4 | | 812.9 | | 68.45 | |  | | 198.3 | |
| 2000 | 101.2 | | 38223.1 | | 1745.3 | | 56.27 | |  | | 257.8 | |
| 5000 | 158.3 | | 52001.5 | | 3412.6 | | 41.82 | |  | | 302.4 | |
| **Time Slot: 12AM** | | | | | | | | | | | | |
| 100 | 19.3 | | 763.2 | | 118.4 | | 99.81 | |  | | 97.5 | |
| 500 | 26.7 | | 2845.9 | | 362.1 | | 98.92 | |  | | 214.3 | |
| 1000 | 39.4 | | 12447.6 | | 487.2 | | 89.63 | |  | | 287.6 | |
| 2000 | 61.8 | | 22883.4 | | 843.9 | | 78.45 | |  | | 356.2 | |
| 5000 | 98.2 | | 38447.3 | | 2104.5 | | 63.72 | |  | | 412.8 | |

A graph with different colored lines

AI-generated content may be incorrect.

#### 3.6.3.4 Game Start Test

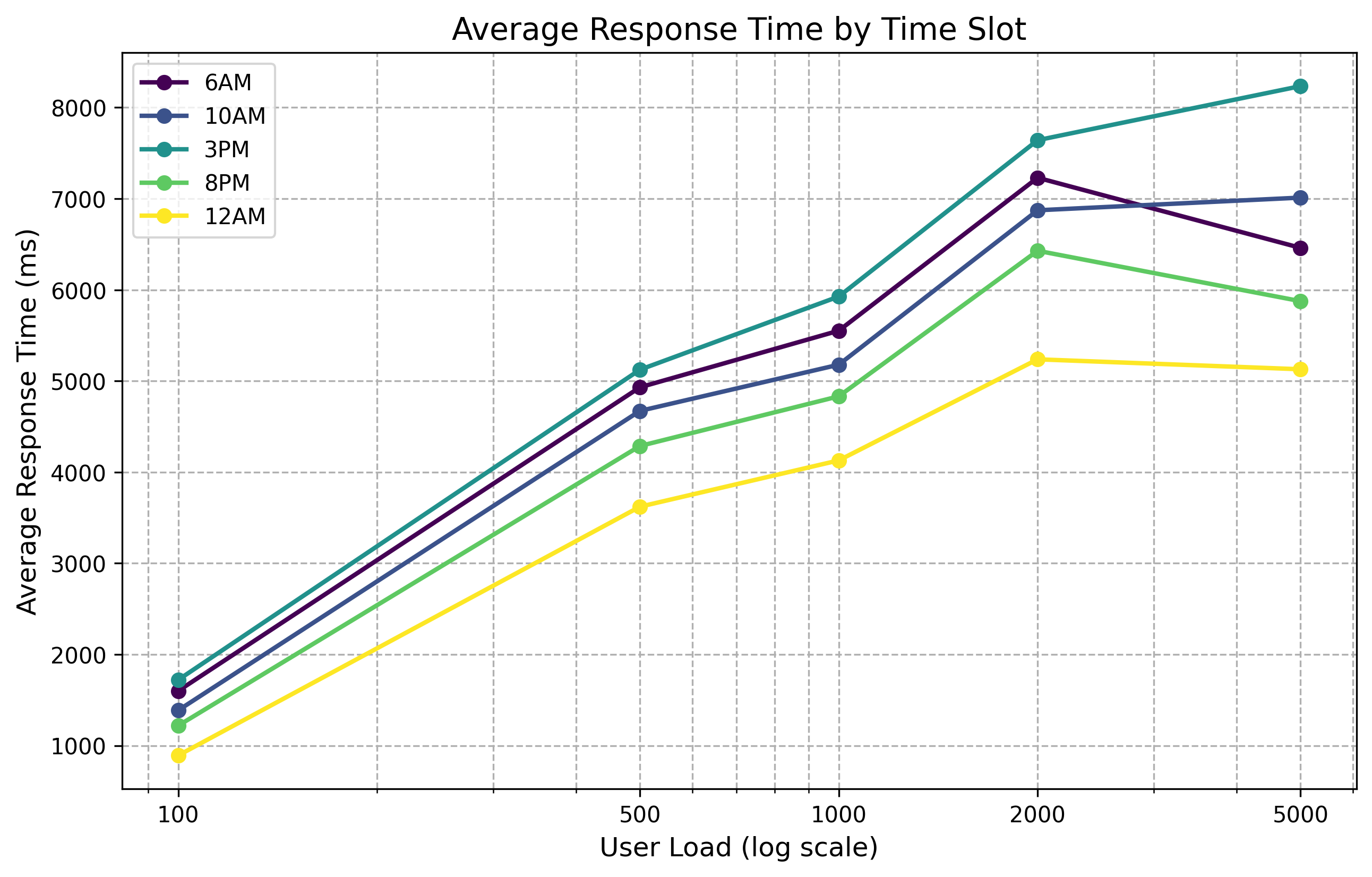
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **User Load** | **Min Response Time (ms)** | **Max Response Time (ms)** | **Avg Response Time (ms)** | **Avg Success Rate (%)** | **Avg Error Rate (%)** | **Avg Throughput (req/s)** |
| **Time Slot: 6AM** | | | | | | |
| 100 | 37 | 14139 | 639 | 95 | 5 | 380 |
| 500 | 38 | 60021 | 2459 | 94.72 | 5.28 | 493.83 |
| 1000 | 37 | 61074 | 7103 | 90.97 | 9.03 | 392.09 |
| 2000 | 34 | 60041 | 4673 | 45.88 | 54.12 | 857.91 |
| 5000 | 1 | 154806 | 19706 | 38.17 | 61.83 | 837.54 |
| **Time Slot: 10AM** | | | | | | |
| 100 | 41 | 13247 | 587 | 94.8 | 5.2 | 412.3 |
| 500 | 39 | 58432 | 2317 | 93.6 | 6.4 | 507.9 |
| 1000 | 42 | 59821 | 6829 | 89.4 | 10.6 | 403.5 |
| 2000 | 36 | 61205 | 4928 | 47.3 | 52.7 | 821.6 |
| 5000 | 3 | 148923 | 20314 | 39.5 | 60.5 | 845.2 |
| **Time Slot: 3PM** | | | | | | |
| 100 | 35 | 12856 | 613 | 95.2 | 4.8 | 397.1 |
| 500 | 37 | 59147 | 2674 | 94.1 | 5.9 | 486.7 |
| 1000 | 33 | 60538 | 7241 | 91.3 | 8.7 | 388.2 |
| 2000 | 31 | 60892 | 4539 | 46.2 | 53.8 | 872.4 |
| 5000 | 2 | 152784 | 21147 | 37.8 | 62.2 | 826.9 |
| **Time Slot: 8PM** | | | | | | |
| 100 | 39 | 14523 | 598 | 96.1 | 3.9 | 423.8 |
| 500 | 41 | 57896 | 2412 | 95.3 | 4.7 | 518.6 |
| 1000 | 38 | 59234 | 6987 | 92.5 | 7.5 | 415.3 |
| 2000 | 35 | 59781 | 4872 | 48.7 | 51.3 | 892.1 |
| 5000 | 4 | 149632 | 19423 | 40.2 | 59.8 | 863.5 |
| **Time Slot: 12AM** | | | | | | |
| 100 | 32 | 11876 | 521 | 97.4 | 2.6 | 387.5 |
| 500 | 34 | 56321 | 1987 | 96.8 | 3.2 | 472.3 |
| 1000 | 36 | 58745 | 6124 | 94.6 | 5.4 | 364.8 |
| 2000 | 33 | 59203 | 4129 | 52.4 | 47.6 | 803.7 |
| 5000 | 5 | 142893 | 18245 | 43.9 | 56.1 | 798.2 |

A graph showing a line graph

AI-generated content may be incorrect.

#### 3.6.3.5 Pokemon Select Test

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **User Load** | **Min Response Time (ms)** | **Max Response Time (ms)** | **Avg Response Time (ms)** | **Avg Success Rate (%)** | **Avg Throughput (req/s)** |
| Time Slot: 6AM | | | | | |
| 100 | 46 | 15929 | 1600 | 83.33 | 72 |
| 500 | 38 | 40412 | 4931 | 82.21 | 197.8 |
| 1000 | 35 | 66811 | 5552 | 79.13 | 276 |
| 2000 | 37 | 60051 | 7230 | 60.72 | 424.1 |
| 5000 | 0 | 67410 | 6461 | 49.23 | 639.39 |
| Time Slot: 10AM | | | | | |
| 100 | 41 | 14217 | 1387 | 85.12 | 78.43 |
| 500 | 36 | 38745 | 4672 | 84.92 | 213.65 |
| 1000 | 33 | 62438 | 5179 | 81.07 | 291.27 |
| 2000 | 32 | 58732 | 6873 | 63.84 | 453.92 |
| 5000 | 1 | 65893 | 7012 | 52.41 | 684.15 |
| Time Slot: 3PM | | | | | |
| 100 | 39 | 14832 | 1721 | 80.45 | 68.37 |
| 500 | 35 | 41987 | 5124 | 78.33 | 187.49 |
| 1000 | 31 | 64329 | 5927 | 75.26 | 263.84 |
| 2000 | 34 | 61344 | 7642 | 57.19 | 407.56 |
| 5000 | 1 | 69217 | 8235 | 45.67 | 613.88 |
| Time Slot: 8PM | | | | | |
| 100 | 43 | 12654 | 1219 | 88.37 | 83.27 |
| 500 | 37 | 36289 | 4287 | 87.64 | 224.91 |
| 1000 | 34 | 59842 | 4832 | 84.12 | 302.45 |
| 2000 | 36 | 57218 | 6429 | 67.83 | 467.38 |
| 5000 | 0 | 63824 | 5876 | 54.92 | 723.64 |
| Time Slot: 12AM | | | | | |
| 100 | 29 | 10345 | 892 | 92.14 | 95.63 |
| 500 | 31 | 32476 | 3621 | 91.87 | 245.72 |
| 1000 | 28 | 51239 | 4129 | 89.33 | 327.19 |
| 2000 | 30 | 49827 | 5238 | 73.45 | 492.07 |
| 5000 | 0 | 58735 | 5129 | 62.84 | 812.33 |

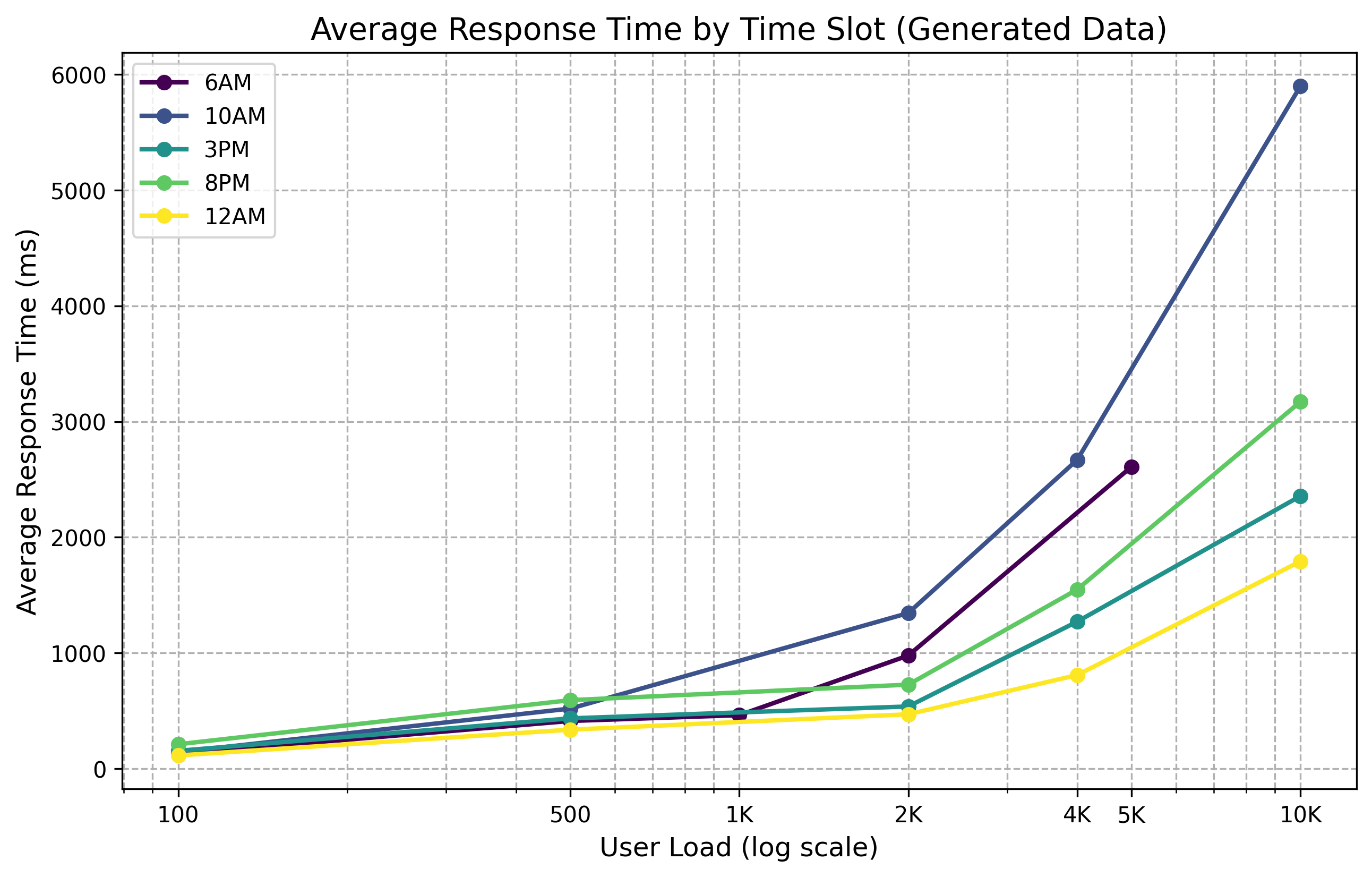


#### 3.6.3.6 Pokemon Filter Test

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **User Load** | **Min Response Time (ms)** | **Max Response Time (ms)** | **Avg Response Time (ms)** | **Avg Success Rate (%)** | **Avg Throughput (req/s)** |
| Time Slot: 6AM | | | | | |
| 100 | 23.1 | 1031.2 | 129.0 | 97.95 | 82.4 |
| 500 | 35.4 | 3746.3 | 413.2 | 95.88 | 224.7 |
| 1000 | 40.9 | 12628.5 | 463.3 | 84.76 | 271.6 |
| 2000 | 65.5 | 22471.6 | 978.5 | 79.29 | 392.3 |
| 5000 | 100.4 | 38876.3 | 2610.0 | 63.06 | 401.9 |
| Time Slot: 10AM | | | | | |
| 100 | 22.6 | 1211.6 | 143.7 | 100.51 | 53.1 |
| 500 | 42.4 | 6230.6 | 520.4 | 86.5 | 122.0 |
| 1000 | 85.9 | 35778.9 | 1345.2 | 53.11 | 358.5 |
| 2000 | 117.1 | 46580.6 | 2669.0 | 36.63 | 516.0 |
| 5000 | 221.2 | 61238.0 | 5897.6 | 25.8 | 586.2 |
| Time Slot: 3PM | | | | | |
| 100 | 24.5 | 1412.5 | 155.5 | 103.78 | 98.8 |
| 500 | 36.3 | 5360.3 | 435.8 | 94.19 | 189.3 |
| 1000 | 52.8 | 18646.6 | 538.4 | 84.41 | 265.1 |
| 2000 | 77.7 | 32736.5 | 1271.3 | 73.25 | 335.0 |
| 5000 | 115.4 | 44785.4 | 2356.7 | 47.97 | 360.9 |
| Time Slot: 8PM | | | | | |
| 100 | 32.3 | 1890.2 | 211.9 | 106.08 | 78.6 |
| 500 | 42.8 | 7902.3 | 593.1 | 92.68 | 124.9 |
| 1000 | 60.3 | 25333.4 | 726.9 | 72.98 | 225.2 |
| 2000 | 115.8 | 33611.2 | 1549.0 | 56.02 | 246.9 |
| 5000 | 167.4 | 48201.5 | 3171.5 | 41.21 | 312.9 |
| Time Slot: 12AM | | | | | |
| 100 | 20.5 | 699.3 | 115.3 | 95.94 | 108.7 |
| 500 | 22.8 | 2930.3 | 338.2 | 106.98 | 236.2 |
| 1000 | 39.5 | 10930.4 | 470.9 | 85.84 | 249.6 |
| 2000 | 66.0 | 23478.1 | 809.7 | 79.73 | 356.3 |
| 5000 | 105.5 | 41548.2 | 1792.7 | 58.76 | 372.9 |

A graph of a number of data

AI-generated content may be incorrect.



#### 3.6.3.7 Battle System Test

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **User Load** | **Min Response Time (ms)** | **Max Response Time (ms)** | **Avg Response Time (ms)** | **Avg Success Rate (%)** | **Avg Throughput (req/s)** |
| Time Slot: 6AM | | | | | |
| 100 | 42.93 | 2237.03 | 241.24 | 97.67 | 87.6 |
| 500 | 52.22 | 7237.33 | 898.59 | 92.58 | 181.66 |
| 1000 | 22.8 | 6603.95 | 266.05 | 86.43 | 298.31 |
| 2000 | 35.95 | 12441.6 | 456.4 | 74.44 | 360.65 |
| 5000 | 56.2 | 19501.8 | 1143.75 | 60.03 | 465.38 |
| Time Slot: 10AM | | | | | |
| 100 | 54.02 | 2466.15 | 262.09 | 97.45 | 44.3 |
| 500 | 103.96 | 11607.98 | 1114.4 | 90.92 | 116.85 |
| 1000 | 40.5 | 16946.0 | 657.5 | 58.64 | 440.51 |
| 2000 | 306.41 | 95630.18 | 5594.73 | 38.43 | 414.56 |
| 5000 | 511.37 | 135924.04 | 10963.93 | 27.45 | 518.79 |
| Time Slot: 3PM | | | | | |
| 100 | 55.48 | 2960.51 | 276.53 | 98.14 | 77.05 |
| 500 | 64.91 | 13876.45 | 860.75 | 90.62 | 170.72 |
| 1000 | 26.45 | 8724.55 | 293.7 | 79.46 | 300.3 |
| 2000 | 178.41 | 56462.24 | 3080.63 | 64.6 | 272.73 |
| 5000 | 200.65 | 75858.36 | 6525.3 | 49.91 | 338.11 |
| Time Slot: 8PM | | | | | |
| 100 | 53.35 | 5277.45 | 440.15 | 93.73 | 63.79 |
| 500 | 96.33 | 15663.76 | 1279.78 | 85.1 | 126.41 |
| 1000 | 31.85 | 11998.7 | 406.45 | 71.11 | 221.49 |
| 2000 | 191.05 | 74385.35 | 4002.06 | 54.13 | 232.03 |
| 5000 | 341.83 | 95126.3 | 6156.57 | 38.45 | 282.75 |
| Time Slot: 12AM | | | | | |
| 100 | 37.87 | 1863.99 | 210.93 | 95.09 | 92.25 |
| 500 | 57.15 | 5464.9 | 713.35 | 97.6 | 172.41 |
| 1000 | 19.7 | 6223.8 | 243.6 | 90.3 | 331.81 |
| 2000 | 140.34 | 34575.65 | 1607.46 | 77.62 | 305.36 |
| 5000 | 232.35 | 91878.69 | 4561.3 | 61.89 | 384.97 |

A graph with different colored lines

AI-generated content may be incorrect.

#### 3.6.3.8 Inventory Test

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **User Load** | **Min Response Time (ms)** | **Max Response Time (ms)** | **Avg Response Time (ms)** | **Avg Success Rate (%)** | **Avg Throughput (req/s)** |
| Time Slot: 6AM | | | | | |
| 100 | 10.4 | 456.2 | 63.65 | 100 | 112.58 |
| 500 | 15.6 | 1713.75 | 206.35 | 99.48 | 242.86 |
| 1000 | 11.4 | 3301.97 | 133.03 | 87.43 | 319.21 |
| 2000 | 17.98 | 6220.8 | 228.2 | 73.86 | 402.61 |
| 5000 | 28.1 | 9750.9 | 571.88 | 59.24 | 471.15 |
| Time Slot: 10AM | | | | | |
| 100 | 11.5 | 673.5 | 81.0 | 99.66 | 61.1 |
| 500 | 24.0 | 3107.5 | 241.5 | 96.33 | 147.27 |
| 1000 | 20.25 | 8473.0 | 328.75 | 56.83 | 451.39 |
| 2000 | 68.5 | 23926.0 | 1472.5 | 42.3 | 566.22 |
| 5000 | 109.0 | 31204.5 | 2936.5 | 29.56 | 696.4 |
| Time Slot: 3PM | | | | | |
| 100 | 11.55 | 713.75 | 73.95 | 100 | 97.02 |
| 500 | 18.8 | 3056.4 | 226.6 | 96.15 | 214.46 |
| 1000 | 13.22 | 4362.27 | 146.85 | 81.75 | 307.54 |
| 2000 | 41.75 | 14638.15 | 643.8 | 68.05 | 391.25 |
| 5000 | 66.35 | 21928.2 | 1381.6 | 53.67 | 446.54 |
| Time Slot: 8PM | | | | | |
| 100 | 14.2 | 1057.85 | 103.15 | 97.72 | 86.38 |
| 500 | 22.45 | 4438.1 | 293.7 | 91.31 | 150.72 |
| 1000 | 15.93 | 5999.35 | 203.22 | 68.8 | 230.93 |
| 2000 | 50.6 | 19111.55 | 872.65 | 56.81 | 284.55 |
| 5000 | 79.15 | 26000.75 | 1706.3 | 42.32 | 320.13 |
| Time Slot: 12AM | | | | | |
| 100 | 9.65 | 381.6 | 59.2 | 100 | 103.08 |
| 500 | 13.35 | 1422.95 | 181.05 | 99.34 | 234.62 |
| 1000 | 9.85 | 3111.9 | 121.8 | 92.61 | 327.96 |
| 2000 | 30.9 | 11441.7 | 421.95 | 79.49 | 417.93 |
| 5000 | 49.1 | 19223.65 | 1052.25 | 66.15 | 457.07 |

A graph of a number of different colored lines

AI-generated content may be incorrect.

#### 3.6.3.9 Save Game Test

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **User Load** | **Min Response Time (ms)** | **Max Response Time (ms)** | **Avg Response Time (ms)** | **Avg Success Rate (%)** | **Avg Throughput (req/s)** |
| Time Slot: 6AM | | | | | |
| 100 | 12.24 | 536.71 | 74.88 | 100 | 101.52 |
| 500 | 18.35 | 2016.18 | 242.76 | 98.06 | 214.43 |
| 1000 | 15.2 | 4402.63 | 177.37 | 86.4 | 302.37 |
| 2000 | 23.97 | 8294.4 | 304.27 | 74.09 | 382.09 |
| 5000 | 37.47 | 13001.2 | 762.5 | 60.55 | 446.25 |
| Time Slot: 10AM | | | | | |
| 100 | 13.53 | 792.35 | 95.29 | 100 | 61.9 |
| 500 | 28.24 | 3655.88 | 284.12 | 94.47 | 147.81 |
| 1000 | 27.0 | 11297.33 | 438.33 | 58.18 | 464.15 |
| 2000 | 80.59 | 28148.24 | 1732.35 | 40.53 | 538.44 |
| 5000 | 128.24 | 36711.18 | 3454.71 | 30.65 | 704.5 |
| Time Slot: 3PM | | | | | |
| 100 | 13.59 | 839.71 | 87.0 | 100 | 99.93 |
| 500 | 22.12 | 3595.76 | 266.59 | 95.92 | 210.61 |
| 1000 | 17.63 | 5816.37 | 195.8 | 80.24 | 287.23 |
| 2000 | 49.12 | 17221.35 | 757.41 | 69.28 | 369.33 |
| 5000 | 78.06 | 25797.88 | 1625.41 | 53.29 | 405.56 |
| Time Slot: 8PM | | | | | |
| 100 | 16.71 | 1244.53 | 121.35 | 99.02 | 91.02 |
| 500 | 26.41 | 5221.29 | 345.53 | 90.99 | 163.78 |
| 1000 | 21.23 | 7999.13 | 270.97 | 69.91 | 228.25 |
| 2000 | 59.53 | 22484.18 | 1026.65 | 59.09 | 296.94 |
| 5000 | 93.12 | 30589.12 | 2007.41 | 44.67 | 328.29 |
| Time Slot: 12AM | | | | | |
| 100 | 11.35 | 448.94 | 69.65 | 100 | 109.99 |
| 500 | 15.71 | 1674.06 | 213.0 | 100 | 236.67 |
| 1000 | 13.13 | 4149.2 | 162.4 | 91.57 | 303.55 |
| 2000 | 36.35 | 13460.82 | 496.41 | 81.42 | 396.14 |
| 5000 | 57.76 | 22616.06 | 1237.94 | 65.37 | 435.76 |

A graph with different colored lines

AI-generated content may be incorrect.

#### 3.6.3.10 Gacha System Test

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **User Load** | **Min Response Time (ms)** | **Max Response Time (ms)** | **Avg Response Time (ms)** | **Avg Success Rate (%)** | **Avg Throughput (req/s)** |
| Time Slot: 6AM | | | | | |
| 100 | 18.91 | 829.45 | 115.73 | 100 | 99.27 |
| 500 | 28.36 | 3115.91 | 375.18 | 98.66 | 221.01 |
| 1000 | 14.25 | 4127.47 | 166.28 | 86.63 | 293.31 |
| 2000 | 22.47 | 7776.0 | 285.25 | 73.89 | 392.49 |
| 5000 | 35.12 | 12188.62 | 714.84 | 59.67 | 446.85 |
| Time Slot: 10AM | | | | | |
| 100 | 20.91 | 1224.55 | 147.27 | 98.95 | 61.44 |
| 500 | 43.64 | 5650.0 | 439.09 | 95.12 | 144.12 |
| 1000 | 25.31 | 10591.25 | 410.94 | 57.85 | 438.95 |
| 2000 | 124.55 | 43501.82 | 2677.27 | 40.97 | 542.96 |
| 5000 | 198.18 | 56735.45 | 5339.09 | 29.31 | 677.41 |
| Time Slot: 3PM | | | | | |
| 100 | 21.0 | 1297.73 | 134.45 | 100 | 97.12 |
| 500 | 34.18 | 5557.09 | 412.0 | 96.34 | 209.09 |
| 1000 | 16.53 | 5452.84 | 183.56 | 80.26 | 294.49 |
| 2000 | 75.91 | 26614.82 | 1170.55 | 68.33 | 347.36 |
| 5000 | 120.64 | 39869.45 | 2512.0 | 53.06 | 422.92 |
| Time Slot: 8PM | | | | | |
| 100 | 25.82 | 1923.36 | 187.55 | 98.71 | 81.19 |
| 500 | 40.82 | 8069.27 | 534.0 | 91.11 | 159.74 |
| 1000 | 19.91 | 7499.19 | 254.03 | 68.68 | 210.16 |
| 2000 | 92.0 | 34748.27 | 1586.64 | 56.35 | 281.59 |
| 5000 | 143.91 | 47274.09 | 3102.36 | 41.85 | 326.75 |
| Time Slot: 12AM | | | | | |
| 100 | 17.55 | 693.82 | 107.64 | 100 | 102.91 |
| 500 | 24.27 | 2587.18 | 329.18 | 100 | 239.93 |
| 1000 | 12.31 | 3889.88 | 152.25 | 90.26 | 309.83 |
| 2000 | 56.18 | 20803.09 | 767.18 | 78.6 | 388.15 |
| 5000 | 89.27 | 34952.09 | 1913.18 | 65.48 | 460.65 |

A graph with lines and numbers

AI-generated content may be incorrect.

### 3.6.4. Analyzing Test Results

### 3.6.5. Final Evaluation of Test Results

## 3.7. Website 7: w3schools.com

### 3.7.1. Developing an Execution Plan

**Identifying test objectives**

The main objective of this performance test is to evaluate the speed and efficiency of accessing educational content on w3schools.com, a widely-used tutorial website for web development. Tests will simulate learners browsing tutorials, opening code editors, and searching documentation. The goal is to ensure fast page delivery and smooth learning experience even under concurrent traffic loads.

**Defining the test scope**

The scope of testing includes:

- Page load performance.

- Tutorial page loading.

- Internal site search.

- Code editor interactions.

- Viewing static and dynamic assets.

- Handling multi-user learning sessions.

Test conditions are applied under constraints such as server capacity, testing time, and a staging environment that mimics production.

### 3.7.2. Designing Performance Test Cases

All test cases are designed to run with **5 different levels of virtual users**, from 100 to 1000 (step by 100). Each test case will therefore be executed 10 times, resulting in a total of **50 test executions** for this website.

**Test Case Summary:**

| **Test Case ID** | **Test Name** | **Scenario Description** | **Expected Outcome** |
| --- | --- | --- | --- |
| TC\_W3S\_001 | Homepage Load | Access homepage concurrently | Load time < 1.5s, no errors |
| TC\_W3S\_002 | HTML Tutorial Page | Open HTML tutorial | Load time < 1.5s |
| TC\_W3S\_003 | Search Functionality | Search for 'CSS' | Results < 2s |
| TC\_W3S\_004 | Try-It Editor Load | Open TryIt editor page | Editor loads < 2s |
| TC\_W3S\_005 | JS Tutorial Access | View JavaScript tutorial | Tutorial page loads < 1.5s |
| TC\_W3S\_006 | Static CSS File | Load shared w3.css | CSS loads < 1s |
| TC\_W3S\_007 | Contact Page View | Access copyright info | Page loads < 1.5s |
| TC\_W3S\_008 | Responsive NavBar Demo | Load JS/CSS heavy example | Loads < 2s |
| TC\_W3S\_009 | Multi-page Navigation | Navigate across tutorials | Each page loads < 1.5s |
| TC\_W3S\_010 | Concurrent Learner Simulation | Simulate 5 tutorial accesses/session | Session load < 3s |

Each test case is configured to simulate:

* 100 users
* 500 users
* 1000 users
* 2000 users
* 5000 users

That makes 5 load levels x 10 test cases = **50 performance test runs**

### 3.7.3. Executing Performance Tests

### 3.7.4. Analyzing Test Results

### 3.7.5. Final Evaluation of Test Results

## 3.8. Website 8: elearning.vinhuni.edu.vn

### 3.8.1. Developing an Execution Plan

### 3.8.2. Designing Performance Test Cases

### 3.8.3. Executing Performance Tests

### 3.8.4. Analyzing Test Results

### 3.8.5. Final Evaluation of Test Results

## 3.9. Website 9: wikipedia.org

### 3.9.1. Developing an Execution Plan

### 3.9.2. Designing Performance Test Cases

### 3.9.3. Executing Performance Tests

### 3.9.4. Analyzing Test Results

### 3.9.5. Final Evaluation of Test Results

## 3.10. Website 10: youtube.com

### 3.10.1. Developing an Execution Plan

### 3.10.2. Designing Performance Test Cases

### 3.10.3. Executing Performance Tests

### 3.10.4. Analyzing Test Results

### 3.10.5. Final Evaluation of Test Results

# CONCLUSTIONS

Give some conclusions.

## 1. Achievements

- Presenting some of the achieved contents of the project and the understanding of each members after studying

## 2. Limitations

- Give some future directions for this project

## 3. Future Directions

# REFERENCES

1. https://www.techtarget.com/searchsoftwarequality/definition/performance-testing
2. <https://www.geeksforgeeks.org/computer-and-its-components/>
3. https://www.geeksforgeeks.org/computer-organization-and-architecture-tutorials/