Automated Verification of Load Test Results in a Continuous Delivery Deployment Pipeline

NIKLAS SUNDBAUM
Automatiserad verifiering av lasttestresultat i en deployment pipeline för kontinuerlig leverans

Automated Verification of Load Test Results in a Continuous Delivery Deployment Pipeline

Sundbaum, Niklas
E-postadress vid KTH: sundbaum@kth.se
Exjobb i: Datalogi
Program: Civileingenjör datateknik
Handledare: Arnborg, Stefan
Examinator: Arnborg, Stefan
Uppdragsgivare: Niklas Sundbaum
Datum: 2015-06-16
Automated Verification of Load Test Results in a Continuous Delivery Deployment Pipeline

Abstract
Continuous delivery is a software development methodology that aims to reduce development cycle time by putting a strong emphasis on automation, quality and rapid feedback. This thesis develops an automated method for detecting performance regressions as part of a continuous delivery deployment pipeline.

The chosen method is based on control charts, a tool commonly used within statistical process control. This method is implemented as part of a continuous delivery deployment pipeline and its ability to detect performance regressions is then evaluated by injecting various performance bottlenecks in a sample application.

The results from this thesis show that using a control chart based approach is a viable option when trying to automate verification of load test results in the context of continuous delivery.

Automatiserad verifiering av lasttestresultat i en deployment pipeline för kontinuerlig leverans

Sammanfattning
Kontinuerlig leverans är en utvecklingsmetodik för mjukvara med målet att reducera ledtid genom att fokusera på automatisering, kvalitet och snabb återkoppling. I det här examensarbetet utvecklas en automatiserad metod för att upptäcka försämringar i prestanda i en deployment pipeline för kontinuerlig leverans.

Den valda metoden baseras på kontroldiagram, ett verktyg som ofta används inom statistisk processkontroll. Metoden implementeras som en del av en deployment pipeline för kontinuerlig leverans och dess förmåga att upptäcka prestandaförsämringar utvärderas genom att olika prestandarelaterade flaskhalsar implementeras i en testapplikation.

Resultaten från arbetet visar att en metod baserad på kontroldiagram är ett tänkbart alternativ för att automatisera verifiering lasttestresultat inom kontinuerlig leverans.
# Table of Content

**INTRODUCTION** ........................................................................................................... 1  
**PROBLEM STATEMENT** .............................................................................................. 2  
  **ASSUMPTIONS AND DELIMITATIONS** .................................................................... 2  
**BACKGROUND** ............................................................................................................ 3  
  **PERFORMANCE TESTING** ......................................................................................... 3  
    **Performance Target Definition** ............................................................................... 4  
  **CONTINUOUS DELIVERY** .......................................................................................... 4  
    **The Deployment Pipeline** ...................................................................................... 5  
    **Load Testing in the Deployment Pipeline** ............................................................... 6  
**PREVIOUS WORK** ......................................................................................................... 7  
**ANALYSIS** ................................................................................................................... 8  
  **CONTROL CHARTS** ................................................................................................... 8  
    **Assumptions on Normally Distributed Data** ......................................................... 10  
    **Requirements Summary** ...................................................................................... 10  
**METHOD** ................................................................................................................... 12  
  **EVALUATION APPROACH** ....................................................................................... 12  
**SOLUTION** .................................................................................................................. 14  
  **SAMPLE APPLICATION** ............................................................................................ 14  
  **LOAD TEST IMPLEMENTATION** .............................................................................. 15  
  **DEPLOYMENT PIPELINE** .......................................................................................... 16  
    **Jenkins Control Chart Plugin** ................................................................................ 17  
  **EVALUATION SETUP** ................................................................................................ 19  
**RESULT** ....................................................................................................................... 21  
  **EXCESSIVE DYNAMIC ALLOCATION** .................................................................... 21  
  **N+1 DATABASE SELECTS** ...................................................................................... 23  
  **SERIALIZED ACCESS** .............................................................................................. 24  
  **MISCONFIGURED CACHE** ....................................................................................... 25  
  **MISSING DATABASE INDEX** .................................................................................... 27  
  **SUMMARY OF RESULTS AND CONCLUSION** ....................................................... 28  
**DISCUSSION** ............................................................................................................... 29  
**REFERENCES** .............................................................................................................. 30
Introduction

In recent years, the concept of continuous delivery has gained a fair amount of attention and traction within the software industry [1]. By putting a strong emphasis on automation, quality and rapid feedback, continuous delivery strives to achieve delivery of “high-quality, valuable software in an efficient, fast and reliable manner” [2].

A central concept within continuous delivery is the deployment pipeline, an automated or semi-automated implementation of an organization’s build, deployment and release processes. This pipeline consists of a series of validation steps, all of which a software artifact must pass before being considered ready for release.

To achieve rapid feedback and to be able to quickly address broken builds, the first validation steps in the pipeline are typically relatively fast, such as compiling or running a suite of in-memory unit tests. As the confidence in this particular version of the software increases, later steps may involve more time-consuming tasks, such as deploying to a live-like test environment for acceptance testing or running a load test to guard against performance related regressions.

Since passing all validation steps means that the software artifact is deemed to have good enough quality to be released to end users, validation of cross-functional requirements, such as performance, must be included in the pipeline.

In the case of load testing, this poses a challenge for many organizations when trying to adopt continuous delivery. Traditionally, load testing has been a time consuming activity involving several manual steps, especially when analyzing various performance related metrics in order to decide if a test has passed or failed.

Thus, traditional approaches to load testing are not compatible with some of the core principles of continuous delivery, such as a high level of automation and rapid feedback. In this thesis, this area will be explored in more detail with an emphasis on automated analysis of load test results in order to decide if a load test has passed its acceptance criteria.
Problem Statement

The goal of this degree project is to develop a method for deciding if a load test has passed or failed, in a way that is compatible with the principles of a continuous delivery deployment pipeline. This means that the method should have the following characteristics:

- Fully automated.
- Provide fast feedback.

In addition to the above, the chosen method must also:

- Provide a high level of transparency, allowing engineers to easily understand why a certain load test passed or failed.
- Allow performance engineers to tune the sensitivity of the pass/fail algorithm.
- Be reliable by generating a low number of false positives and negatives.

This leads to the following problem statement:

*How can verification of a load test's acceptance criteria be automated in a way that is appropriate for use within a continuous delivery deployment pipeline?*

Assumptions and Delimitations

For the scope of this thesis, it is assumed that the application under test is a web based client-server application. Hence, applications such as embedded systems, native mobile or desktop applications will not be considered.

Furthermore, this thesis will not address the challenge of identifying the software component responsible for any detected performance degradations, nor will it provide guidance on how to select an appropriate set of performance counters to base the test verification on.
**Background**

In this chapter, the underlying concepts of continuous delivery and performance testing that this thesis builds on will be described. Additionally, a brief overview of previous academic work related to this thesis will be provided.

**Performance Testing**

Performance testing in the context of software engineering refers to the process of determining how a software system behaves in terms of responsiveness, throughput, reliability and scalability when exposed to a particular workload [3].

On a high level, the performance testing process consists of the following steps [4]:

1. *Nonfunctional Requirements Capture*. In this step, the scope of the test effort is determined, the performance target is identified and key test use-cases are identified. Other practical considerations such as requirements for the test environment, tool selection, timelines and deliverables in terms of test reports are included in this step.

2. *Performance Test Environment Build*. In this step, the performance test environment is setup, links to external systems are created or stubbed, the system under test is deployed and any tools needed for testing, such as key performance indicator (KPI) monitoring, are installed and configured.

3. *Use-Case Scripting*. In this step, the use-cases identified in step 1 are scripted and tested. As part of this, checkpoints for gathering and monitoring response times are implemented at relevant places in the scripts.

4. *Performance Test Scenario Build*. In this step, the details of the test execution are decided on. This includes items such as the types of tests to run, number of users (also commonly referred to as virtual users) to simulate for each scenario, ramp-up times and any delays (to simulate user “think time”) between different use-cases, test execution time, runtime monitoring and KPI gathering.

5. *Performance Test Execution*. In this step, the actual tests are executed and monitored.

6. *Post-Test Analysis and Reporting*. In this step, the data generated by the test execution is gathered and analyzed and the results of the tests are documented in a test report. Finally, using the performance targets identified in step 1 as input, the test run is deemed as either a success or a failure.

For this thesis, the focus will primarily be on step 6, especially with regards to determining whether a test succeeded or not, and to some extent to the performance target definition in step 1.

Performance tests are usually categorized into different sub-types, depending on the primary purpose of the test and how the test is setup and structured. The exact classification and terminology differs slightly between different literature ([2], [3], [4] and [5]), but at least the following broad categories are commonly used:

- **Load test**. The system under test is exposed to the target load in order to verify that the performance targets are met under a production-like load.
- **Stress test**. The load on the system under test is ramped up until the application or supporting infrastructure breaks. The purpose of this test is to determine the capacity threshold for the system.
Background

- **Soak test.** In this type of test, the system under test is exposed to a relatively moderate load for an extended period of time. The primary purpose of this test is to reveal if performance degrades over time, for example due to memory leaks or other stability related problems.

- **Scalability test.** This test is performed to determine how capacity change as more resources, such as servers, memory or CPUs, are added to the system under test.

- **Isolation test.** This type of test is performed to investigate identified problems in more detail. It is usually set up to execute only the particular use case or test execution that triggered the performance issue.

This thesis primarily focuses on load testing, but may be applicable to other types of performance testing as well.

**Performance Target Definition**

In order to set an acceptance criteria for a performance test, a clear and well-defined performance target must be established [4]. These performance targets are usually defined in terms of one or more of the following metrics:

- **Availability.** The amount of time that the key features of an application are available to the user.

- **Response time.** The time it takes to process a single business transaction within the system, e.g. to respond to a user request. This may or may not include the time taken to render the response on the client side.

- **Throughput.** The total number of transactions that a system can handle per unit of time.

- **Concurrency.** In the context of performance testing, this is often defined as the number of active simulated application users.

- **Network utilization.** The amount of data sent over the network.

- **Server utilization.** For example CPU utilization, memory usage and disk input/output.

A performance **baseline** consists of data, expressed in terms of the performance metrics above, collected from a specific test execution [3]. This baseline can later be used to compare different executions of the same performance test in order to evaluate whether changes to the system under test have a positive or negative impact on performance. The process of comparing a test execution against a baseline is called a **benchmark**.

It is important to keep in mind that all changes to a system under test, except for those explicitly being tested, will invalidate the baseline. As an example, consider the case where a test suite is executed against a hardware configuration that is different from the one used for the baseline. In this scenario, the baseline will not provide any guidance on whether the performance of the actual software being tested has changed. It can however be used to evaluate if the hardware setup used for the test is better (from a performance point of view) compared to the one used for the baseline test.

**Continuous Delivery**

Continuous delivery is a set of software engineering principles for delivering software that aim to reduce cycle time while still ensuring that the software exhibits an appropriate level of quality [2]. In this context, cycle time is defined as the time
between deciding to make a change to the software system and that change being available to end-users in a production environment.

In recent years, continuous delivery has been adopted by a wide range of companies, such as The Guardian [6], Amazon [7], Google [8], Etsy [9] and Spotify [10], and studies suggest that adopting continuous delivery practices has a strong positive correlation with IT and organizational performance [11], [12].

To achieve the goals of short cycle time and high quality, continuous delivery puts a strong focus on automation, both when it comes to testing and processes. The driving forces here are to achieve repeatability of the build, deploy, test and release processes, avoid errors due to the human factor and to get traceability so that it is possible to review precisely what has been done in a particular step.

Emphasis is also put on making frequent production releases of the software, ensuring that the delta between releases is small. This decreases the risk associated with a release by making errors easier to pinpoint and simplifying roll back procedures.

According to continuous delivery, rapid feedback is an essential component needed to succeed with frequent and automated releases. More specifically:

- Any change needs to trigger the feedback process (see the next chapter).
- Feedback should be available as soon as possible.

The Deployment Pipeline

In continuous delivery, everything is kept under version control and every check-in to version control leads to a potential release. However, before a particular version of the software can be released to production, the quality of the software must be assessed. Only if the quality meets the agreed standards, the software can be released to end-users.

In order to realize this quality assessment, continuous delivery stipulates the implementation of a deployment pipeline: a semi or fully automated implementation of an organization’s build, deploy, test and release processes. Every change of code, configuration and/or infrastructure is propagated through this pipeline with the objective to eliminate defective or broken release candidates as soon as possible.

At a high level, the deployment pipeline consists of a series of tests in different stages that the software must pass before being propagated to the next stage. Some of these stages may be manual, and the decision whether to propagate a release candidate to an upstream stage may be manual, but in general as many stages and tests as possible should be fully automated.

To achieve a rapid feedback cycle, the first stages of the deployment pipeline focus on relatively fast tests, such as compilation or in-memory unit tests. However, as the release candidate passes each stage, the confidence in the production readiness of this particular version grows, which means that it is motivated to increase the amount of resources and time that are spent on assessing quality. Thus, later stages of the pipeline may consist of relatively long running tests, such as automated acceptance tests, user acceptance tests and load tests, against a live-like test environment.

---

1 Note that it is still a business decision whether a particular version of the software will actually be released. This is one of the key differences between continuous delivery and continuous deployment [32].
To illustrate this concept, consider the hypothetical (but fairly typical) deployment pipeline in Figure 1 below.

![Diagram of deployment pipeline stages](image)

**Figure 1. Example deployment pipeline stages.**

The *commit stage* is the first stage of the pipeline and consequently the first thing that occurs after a developer has checked in code to version control. In this stage, code is compiled (if applicable) and a set of unit tests and other fast running tests are executed. Next, static code analysis tools are run against the code base to verify any agreed coding conventions and gather code level metrics such as test coverage, code duplication, cyclomatic complexity and coupling. Finally, the code is assembled into a deployable artifact that can be used to deploy the code to a test or production environment in subsequent steps.

Upon successful completion of all steps in the commit stage, the *acceptance test stage* is triggered. Here, the software artifact created in the previous step is deployed to a test environment against which automated acceptance tests are carried out.

Next up is the *user acceptance test stage*. The software is deployed to a test environment in a similar fashion to the previous stage, but in this stage the tests carried out are manual instead of automated. Examples of tests performed in stage include exploratory testing, acceptance testing and usability testing.

The final test stage in this example is the *load test stage* where the system’s capacity is verified and performance related regressions are caught. This stage, which is explored in more detail in the next section, is the stage that this thesis concentrates on.

Finally, after successful completion of all stages, the software may be released to production as part of the *release stage*.

**Load Testing in the Deployment Pipeline**

Adding an automated load testing stage to the deployment pipeline can be a challenge since such tests are typically more fragile and complex compared to unit or functional tests. However, since non-functional requirements related to performance and capacity constitute a significant risk to software projects [2] [4] [13] [14], it is important to get feedback as quickly as possible after performance related issues have been introduced. Therefore, at least in some contexts, the extra cost of adding a load test stage to the deployment pipeline is motivated.

Another difficulty related to this is how to define the success criteria in a way that can be easily automated [2]. A criterion too generous will make the tests less useful, since they may miss actual performance issues, while a criterion too restrictive might result in false positives due to various types of intermittent failures.

When it comes to automation of performance tests, Humble and Farley [2] list a few characteristics relevant to this thesis, that performance tests in a deployment pipeline should exhibit:

- They should have a predefined success criterion.
- They should be short in duration to get feedback as soon as possible.
- They should be robust.
Previous Work

Previous academic work relevant to this thesis include Malik et al. [15], where four different methods for automatic detection of performance regressions during load tests were evaluated. From a high level, their approach consists of two steps. First, a large set of performance metrics are reduced into a smaller, more comprehensible, set called a performance signature. Then, the performance signature is compared with a baseline signature to detect any performance deviations.

For performance deviation detection, i.e. the area relevant to this thesis, control charts, principal component analysis and a supervised “WRAPPER” approach was used. The results show that the supervised approach gave the best results, but all approaches were able to detect performance deviations in the majority of cases.

In a similar fashion, Foo et al. [16] extracted performance signatures from a repository of historical performance test data. To form the signatures, metrics are discretized into high/medium/low buckets and then data-mining techniques are used to extract frequently observed metric correlations into a set of association rules, such as:

medium request rate AND medium throughput EQUALS medium cpu utilization

Any violations of these rules in subsequent tests are used as an indicator of a performance regression.

Jiang et al. [17] analyzed application execution logs generated during performance tests to identify and abstract use cases into sequences of events. The performance of each of these events is calculated and compared to a baseline test using statistical analysis and any deviations between the baseline and the target test are assembled into a report.

Nguyen et al. [18] used control charts to compare metrics from different test runs and flag performance deviations. First, an initial performance test is used to generate a baseline control chart based on application response times. This chart includes an upper control limit (e.g. at the 90th percentile) and a lower control limit (e.g. the 10th percentile). Subsequent tests are then compared to this baseline by calculating a violation ratio based on the number of data points that are outside of the upper and lower control limits. A high violation ratio is used as an indicator of a performance regression.
Analysis

To recap, the goal of this degree project is to implement a step in a deployment pipeline that automates the verification of a load test’s acceptance criteria. In the “Problem Statement” chapter, the following was listed as requirements for the chosen solution:

1. Fully automated.
2. Fast feedback.
3. High level of transparency, allowing engineers to easily understand why a certain load test passed or failed.
4. Allow performance engineers to tune the sensitivity of the pass/fail algorithm.
5. Reliable, by generating a low number of false positives and negatives.

Perhaps the most simple and straightforward approach would be to base the acceptance criteria exclusively on the total average response time for a load test execution. However, while this might fulfill requirements one to four above, it would unfortunately fail to satisfy the fifth requirement under some circumstances.

To illustrate this, consider the case where a baseline load test execution has three response time data points, 145, 155 and 150 milliseconds, resulting in a total average of 150 milliseconds. Based on this baseline, the performance engineer sets the acceptance criteria to 160 milliseconds to give some room for outliers.

A new test is performed, yielding response times of 145, 155 and 165 milliseconds, which results in an average of 155, i.e. well within the established limit.

Now, a third test is performed, which gives the exact same total average as the second test, but this time with individual response times of 100, 185 and 180 milliseconds.

Clearly, the second and the third tests are very different in their response time distribution, and one might even argue that the third test should be considered a failure since two thirds of the data points are above the defined limit. However, they still produce the same average. Hence, a slightly more refined algorithm, which considers variance as well as averages, is needed to fulfill all the stated requirements.

By evaluating the academic work in this area (see the “Previous Work” chapter), the control chart approach outlined by [18] seems like a good fit for this context. Many of the other strategies rely on fairly advanced statistical methods or data mining concepts, which may be a challenge to explain and illustrate to a layman performance engineer and thus failing to satisfy the third requirement.

Control Charts

The concept of control charts\(^2\) originates from work performed by Walter A. Shewhart at Bell Labs in the 1920s. While originally used as a means to monitor the quality of components in telephony transmission systems, the technique is nowadays widely spread and one of key tools used in statistical process control [19].

The primary goal of control charts is to detect variation in a process that is not randomly caused [20]. The output of most, if not all, processes exhibit some degree of variation as part of their natural operating state. This is often referred to as chance causes or common causes. Other, usually larger, amounts of variation come from external sources.

\(^2\) Sometimes referred to as Shewhart charts or process-behavior charts.
such as bad raw material or improperly tuned machines. This is referred to as assignable causes or special causes. When a process is subject to special causes, it is said to be out-of-control and it is this scenario that control charts try to identify.

To construct a control chart, sample measurements of a selected quality metric are taken from a process at different points in time and plotted on a graph. In addition to this, a horizontal center line (CL) is drawn to visualize the average value of the metric. Above and below the center line, two additional lines known as upper control limit (UCL) and lower control limit (LCL) are added. These lines represent the boundaries for an in-control process and are chosen in such a way that data points outside of the control limits due to common causes are highly unlikely. As long as the data points fall between the two control limits, the process is said to be in-control, while metrics outside the boundaries indicate an out-of-control process that may require some kind of corrective action.

At each sample point, usually more than one sample is gathered to ensure better process deviation detection. This is commonly referred to as sample size.

An example of a control chart is illustrated in Figure 2 below. In this example, the corresponding process is in-control for the first eight observations while the ninth sample falls outside the control limits, which may indicate an out-of-control process.

![Figure 2. Example control chart.](image)

To determine the placement of the upper and lower control limits in the general case, the following formulas are used [20]:

\[
UCL = \mu_w + L\sigma_w \\
CL = \mu_w \\
LCL = \mu_w - L\sigma_w
\]

_Equation 1. Control chart limits._

Here, \( w \) is the sample metric being measured, \( \mu_w \) is the mean of \( w \) and \( \sigma_w \) is the standard deviation. \( L \) is a constant that defines the distance of the control limits from the center line. Typically, \( L \) is assigned a value of 3, which in that case is referred to as three-sigma control limits.

It is worth noting that there exists a number of different types of control charts [19][20]. Which one to use for a certain situation depends on factors such as the type of metric being measured (e.g. variables or attributes), number of samples taken and whether...
means or variability is being monitored. A few of the most common ones are briefly mentioned below:

- An \( \bar{x} \) chart monitors process mean.
- An \( R \) chart monitors process variation by plotting the range of a process metric.
- An \( s \) chart monitors process variation by plotting the standard deviation of a process metric.
- An *individuals control chart* monitors process mean and moving range in two different charts. As opposed to the \( \bar{x} \) and \( R \) chart, an individuals control chart is used when one “unit” at the time is inspected, something that may be required for example when production is so slow that it is impractical to wait for more than one unit arriving (i.e. to use a sample size > 1) before analysis is performed.

As illustrated by “Equation 1”, control chart limits are based on the mean and standard deviation of the output population being measured. In traditional manufacturing industries, where control charts were originally most commonly used, it is often unpractical and expensive to calculate the actual population mean and standard deviation, since that would involve taking measurements from every single unit produced. Many of the control chart types described in literature are therefore designed around various strategies for estimating mean and standard deviation based on samples from the actual population.

However, in an automated load testing context where all the data points, i.e. the entire population, for a specific metric usually are readily available in digital form, the concept of sampling makes less sense since it would involve throwing away gathered data. Hence, for the context of this thesis, the formulas in “Equation 1” can be used as-is, with calculations for mean \( \mu \) and standard deviation \( \sigma \) for a population of size \( n \) following the standard formulas [21]:

\[
\mu = \frac{\sum_{i=1}^{n} x_i}{n}
\]

\[
\sigma = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \mu)^2}{n}}
\]

*Equation 2. Population mean and standard deviation.*

**Assumptions on Normally Distributed Data**

There is a lack of consensus within the statistical process control community as to whether the underlying probability distribution of the data being used in a control chart has to be normally distributed or not [22].

This thesis will take the standpoint of Wheeler [23], who argues that control charts using three-sigma limits work well with a wide variety of probability distributions. Wheeler backs his argument with own research of a number of different probability models as well as Shewhart’s original work.

**Requirements Summary**

Based on the key characteristics of control charts described in this chapter, it is clear that using control charts could potentially fulfill all the necessary requirements:

1. *Fully automated.* All the steps in the control chart algorithm are highly suitable for automation.
2. *Fast feedback.* There is nothing inherent to the control chart concept that stipulates a large number of collected data points, so fast feedback should be possible to achieve.

3. *High level of transparency.* Control charts are very visual to their nature and there is a direct mapping between measured load test metrics, such as response times, and the corresponding visualized control chart. Additionally, the underlying concept is relatively simple and intuitive to explain, as noted by [18]. This has also been confirmed by the author of this thesis, through various informal discussions with colleagues.

4. *Tunable sensitivity.* The sensitivity of the pass/fail algorithm can be tuned by modifying the allowed violation rate, i.e. the number of data points the fall outside the control limits.

5. *Reliable.* Control charts are widely used within various different industries and considered one of the core tenants of statistical process control [20]. The work performed by Nguyen et al. [18] shows that this technique is applicable in a software load testing context.
Method

In order to evaluate the chosen approach in a continuous delivery context, a sample deployment pipeline is needed. To simulate a somewhat realistic setup, this pipeline should be fully automated and consist of at least the following steps (see “Figure 3”):

1. Build, compile and run unit tests for the system under test.
2. Deploy the compiled artifact to a test environment.
3. Execute a load test against the test environment.
4. Verify the load test result against its acceptance criteria using a control chart based approach.

Additionally, as a prerequisite before the pipeline can be executed, a load test baseline should be established to define the load test acceptance criteria for step 4. To be compatible with the chosen approach, this baseline must be expressed in the form of a control chart with appropriate control limits. It must also be possible to tune the sensitivity of the acceptance criteria by specifying an allowed violation rate that defines the maximum number of data points that are allowed to fall outside of the control limits before the test is considered a failure.

When verifying load tests against a baseline, several different quality metrics, such as response times, CPU utilization, memory consumption and disk I/O are usually considered. However, to keep things simple this work will only consider response times. As there is nothing inherently special with response times compared to most other performance metrics, it is assumed that the results of this method can be applied to other metrics as well.

Evaluation Approach

To evaluate the chosen method, the baseline acceptance criteria will first be established by running a load test against a known good version of the system under test. Based on the metrics generated by this load test, response time mean and standard deviation values can be calculated from the gathered metrics. Three-sigma control limits can then be set up using the formulas in “Equation 1”, with $L = 3$.

Next, various common performance bottlenecks will be injected into the system under test to simulate different performance related regressions. Based on [24] and the author’s own experience, the following scenarios are chosen to represent common performance anti-patterns:

- *Excessive dynamic allocation.* In this scenario, an object that is expensive to create is instantiated and then destroyed every time it is used. Common
examples of this are objects used to connect to other processes such as databases or services across a network boundary.

- **N+1 database selects.** In this anti-pattern, the software needs to retrieve a parent-child relationship from a database. Instead of implementing this using a SQL join statement, the program logic first fetches all the parent entities using one database query and then, for each parent, issues a new query to get the corresponding child entity.

- **Missing database index.** Here, a database table is missing an index, making queries against that table inefficient.

- **Serialized access.** In this situation, a multi-threaded program execution is serialized at a point in the code using some kind of locking mechanism.

- **Misconfigured cache.** Caching of a resource that is expensive to retrieve is misconfigured in a way that causes the resource to be retrieved more often than intended.

By running load tests against each of the above scenarios and evaluating the corresponding result in a control chart using the baseline’s control limits, the method’s accuracy in detecting such issues will be measured.
Solution

To implement the method outlined in the previous chapter, a number of different components and frameworks are needed. In the next sections, these components, along with implementation details relevant to the core of this thesis, are described in more detail.

Sample Application

The open-source application Spring PetClinic [25] was used to act as the system under test. Spring PetClinic is a Java based sample application that aims to serve as an example of how to build applications using the Spring Framework toolkit [26]. Spring PetClinic is a web based application that is accessed through a web browser. It simulates an administrative system for a pet clinic where veterinarians can perform tasks related to clients and their pets, e.g., add and search for clients, assign pets to clients and update information regarding visits and treatments.

From a technical point of view, the application is built around a three-layer architecture with presentation, business and persistence logic separated into different modules. It uses a relational database for persistence, which can either be in memory, using a HSQLDB database, or running in a separate process, using a MySQL database. For the purpose of this work, the MySQL option was used. The high-level architecture of the sample application is illustrated in “Figure 4” below.

Since the Spring PetClinic application is published under an open-source license, the application’s source code is readily available and possible to extend or modify. After checking out the source code from the project’s central source code repository, five local repository branches were created; one for each performance bottleneck to simulate, as defined in the previous chapter. On each branch, the corresponding performance anti-pattern was implemented. The implementation details of these changes are summarized in “Table 1” below.
### Performance anti-patterns

<table>
<thead>
<tr>
<th>Performance anti-pattern</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Excessive dynamic allocation</td>
<td>Instantiating new database connections each time the application needs to access the database instead of utilizing a database connection pool.</td>
</tr>
<tr>
<td>N+1 database selects</td>
<td>Loading pet owners from the database in one query and then issuing a separate database query to load each owner’s pets, when searching for pet owners.</td>
</tr>
<tr>
<td>Serialized access</td>
<td>Synchronizing access to the list owner view.</td>
</tr>
<tr>
<td>Misconfigured cache</td>
<td>Removing caching from the list veterinaries view.</td>
</tr>
<tr>
<td>Missing database index</td>
<td>Removing the database index on owner surname that is utilized when searching for an owner.</td>
</tr>
</tbody>
</table>

*Table 1. Performance anti-patterns implementations.*

To ensure that the load tests are run under somewhat realistic conditions, the script that populates the database with initial data was also modified to contain larger amounts of owners, pets and visits. The original application contains approximately ten entries each for these entities and this was increased by roughly a factor of 100.

### Load Test Implementation

For load test use-case scripting, load generation and metrics collection, the open-source framework Gatling [27] was utilized. Gatling is a Scala based framework for building, launching and analyzing load tests.

By exploring a running instance of the chosen Spring PetClinic application through a browser, the following use-cases were identified:

<table>
<thead>
<tr>
<th>Use-case name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Home</td>
<td>Access the landing page of the application.</td>
</tr>
<tr>
<td>List vets</td>
<td>Lists all veterinarians in the system.</td>
</tr>
<tr>
<td>Search owners</td>
<td>Searches for a pet owner by last name.</td>
</tr>
<tr>
<td>View owner</td>
<td>Views information about an owner retrieved in the “Search owners” use-case.</td>
</tr>
<tr>
<td>Add pet</td>
<td>Adds a pet to an owner</td>
</tr>
<tr>
<td>Add visit</td>
<td>Adds information about an owner’s visit.</td>
</tr>
<tr>
<td>Edit owner</td>
<td>Edits information, such as name and address, about an owner.</td>
</tr>
<tr>
<td>Add owner</td>
<td>Creates a new owner.</td>
</tr>
</tbody>
</table>

*Table 2. Key load test use-cases.*

These use-cases were then implemented in Gatling and tested against the sample application to ensure that they work as intended. Between each request to the server, a randomly selected pause between one and three seconds was added for the corresponding virtual user to simulate user “think time”.

To simplify the load test scripts, only the raw html content, and thus no static resources such as images, cascading style sheets or JavaScript files, was fetched.

The load test suite was also implemented to support a few configuration options, listed in "Table 3" below.
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of users</td>
<td>Total number of virtual users to simulate.</td>
</tr>
<tr>
<td>Ramp up time</td>
<td>Over this time, users are ramped up at a constant rate until the maximum number of users (as specified by the “Number of users” parameter) is reached. This ensures that user activities are spread out in time and that the system under test is not overwhelmed at load test startup.</td>
</tr>
<tr>
<td>Duration</td>
<td>Duration of entire load test.</td>
</tr>
</tbody>
</table>

Table 3. Load test configuration options.

Different configuration options were primarily used during testing and tuning of the load test process. During the actual evaluation load tests, all test executions were launched with the exact same set of configuration parameters.

**Deployment Pipeline**

To define and orchestrate the flow of a deployment pipeline, like the one illustrated in “Figure 3”, a continuous integration server is commonly used. For the work of this thesis, the open-source continuous integration server Jenkins [28] was utilized.

Jenkins is built on a plugin architecture, which makes it relatively easy to extend and customize its behavior by utilizing existing plugins or writing custom ones.

From a conceptual point of view, a deployment pipeline in Jenkins consists of one or more jobs, where each job consists of a number of steps. To implement the deployment pipeline, two different jobs were created, as illustrated in “Figure 5”.

![Figure 5. Implemented deployment pipeline.](image)

The first job compiles the sample application, runs all unit tests and creates a software artifact that can be deployed in an application server. Finally, assuming the previous steps were successful, this job deploys the application to a running Tomcat [29] application server instance and restarts the Tomcat server.

Upon successful execution of the first Jenkins job, a second job is automatically triggered to kickoff a Gatling load test against the running Tomcat instance.

To give both the Tomcat application server instance and the sample application some time to “warm up” and perform task such as cache population and JIT optimizations, a short warm-up load test is run before the main load test is started. This test does not generate any reports or metrics.
Once the primary load test has completed, the load test result is evaluated and plotted on a control chart with control limits specified by the baseline test. This required a custom Jenkins control chart plugin to be developed, which is described below.

**Jenkins Control Chart Plugin**

While the majority of the logic in the deployment pipeline was implemented using existing Jenkins plugins, there exists no publically available plugin for verifying load test results using control charts (at least not to the authors knowledge). Hence, as part of this work, a Jenkins control chart plugin had to be developed.

The control chart plugin is intended to run as a Jenkins post-build step after a Gatling load test has been performed and the corresponding response time metrics have been stored.

Response time metrics from a Gatling load test are stored in a file called `simulation.log`, which lists data for each executed request on the following format, where tabs separate individual fields and the newline character separates requests [30]:

```
[Load Test Name] [Test User Id] REQUEST [Test Use-Case] [Request Start] [Request End] [Response Start] [Response End] [Status Code]
```

The control chart plugin starts by locating the `simulation.log` file somewhere under the currently executing job’s root directory. It then parses the file line by line and looks for lines where the third field has a value equal to `REQUEST`\(^3\). On a match, it calculates the response time by subtracting the “request start” field from the “response end” field. This value is then stored in a map like data structured using the “test use-case” field as key. This means that metrics for each use-case will be separated and tracked individually.

During this process, response time mean and standard deviation is also calculated for each use-case. Additionally, if there is a baseline specified, the violation rate against that baseline is tracked. Since this plugin only deals with response times, where larger values are worse, it only considers the upper control limit when calculating violation rates.

Finally, the plugin iterates over all use-cases and fails the build if any use-case has a violation rate greater than the specified threshold. If no baseline is specified, this last step is omitted.

The control chart plugin also contains a visual reporting feature that allows a Jenkins user to inspect metrics for each use-case in a table and to see a graphical representation of the control charts, where each data point is plotted in a chart together with horizontal marker lines for the mean and control limits.

“Figure 6” and “Figure 7” below show two screen-shots from the plugin’s user interface.

---

\(^3\) In addition to request metrics, Gatling also stores other types of events in this file. The third field identifies the type of event. Therefore, this check is needed.
Figure 6. Table view of load test results.

Figure 7. Load test control chart.

Configuration

The control chart plugin offers a few job specific configuration options that are available from the standard Jenkins job configuration view, see “Figure 8”.

First, a violation rate can be specified as a maximum percentage of requests that are allowed to fall above the baseline’s upper control limit before the corresponding test is considered a failure. This is currently specified globally for all use-cases in a specified load test.

Secondly, a baseline can be defined by providing mean and standard deviation values for each individual use-case. From these, the control chart plugin will then calculate proper control limits.

It is possible to define baseline values for only a subset (or none) of the use-cases that constitute a certain load test.
Evaluation Setup

To evaluate the chosen approach against the various performance bottlenecks listed in “Table 1”, the deployment pipeline was first implemented in Jenkins using the configuration described in previous chapters.

Next, two servers were configured to host the Spring PetClinic sample application; one application server and one database server. These were setup in a virtualized environment using virtualization software from VMware [31]. The specifications for the two application servers and the Jenkins server are listed in “Table 4” below.

<table>
<thead>
<tr>
<th></th>
<th>Application Server</th>
<th>Database Server</th>
<th>Jenkins Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System</td>
<td>Debian 7.0</td>
<td>Debian 7.0</td>
<td>OSX 10.9</td>
</tr>
<tr>
<td>CPU</td>
<td>Intel Xeon 2.30 GHz</td>
<td>Intel Xeon 2.30 GHz</td>
<td>Intel Core i7 2.2 GHz</td>
</tr>
<tr>
<td>CPU cores</td>
<td>4</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>Memory</td>
<td>4 GB</td>
<td>10 GB</td>
<td>16 GB</td>
</tr>
</tbody>
</table>

Table 4. Test server specifications.

The load test itself was configured to use 250 simulated users and run for approximately six minutes, including a ramp-up time of 60 seconds. As mentioned in the “Deployment Pipeline” chapter, each test was preceded by a Tomcat restart, which also

---

4 Gatling was configured let each simulated user complete its current tasks once the test reached its time limit, hence the approximate load test length.
reset the database and populated it with initial test data. Before the main load test was run, a warm-up load test was performed using a load of 20 simulated users for 60 seconds.

To verify that the results from the load tests were consistent, each load test was repeated three times against the same software configuration, using identical setup procedures. The first set of load tests were run to establish a baseline to use for further testing and evaluation. Next, a suite of “control tests” was executed against the exact same configuration (i.e. without any injected performance bottlenecks) to ensure that this did not produce any false positives.

After that, a set of load tests was run against each of the performance anti-patterns listed in “Table 1” and results were recorded and evaluated. The outcome of these tests will be described in the next chapter.
Result

To establish the baseline, the deployment pipeline was triggered for the known good version of the sample application and mean and standard deviation metrics for each load test use-case was collected. This was repeated three times, and for each use case, the test yielding the highest upper control limit was chosen, resulting in the baseline below (“Table 5”).

<table>
<thead>
<tr>
<th>Use-case</th>
<th>Mean (ms)</th>
<th>Standard Deviation (ms)</th>
<th>Upper Control Limit (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Home</td>
<td>9.1</td>
<td>5.6</td>
<td>25.9</td>
</tr>
<tr>
<td>Vets</td>
<td>12.4</td>
<td>3.7</td>
<td>23.5</td>
</tr>
<tr>
<td>Find</td>
<td>131.1</td>
<td>103.0</td>
<td>440.0</td>
</tr>
<tr>
<td>View Owner</td>
<td>19.8</td>
<td>8.6</td>
<td>45.8</td>
</tr>
<tr>
<td>Add Pet</td>
<td>24.4</td>
<td>9.3</td>
<td>52.4</td>
</tr>
<tr>
<td>Add Visit</td>
<td>21.8</td>
<td>8.1</td>
<td>46.1</td>
</tr>
<tr>
<td>Edit Owner</td>
<td>23.3</td>
<td>7.4</td>
<td>45.6</td>
</tr>
<tr>
<td>Add Owner</td>
<td>15.5</td>
<td>10.4</td>
<td>46.6</td>
</tr>
</tbody>
</table>

Table 5. Load test baseline.

To ensure a stable baseline, a set of control load tests against the known good version of the application was run three times using the baseline as input. For each of these three test runs, violation rates were computed. As illustrated by “Table 6” below, the maximum violation rate for these tests were 1.8% (see Control 3, “Edit Owner”). Based on that number, a violation rate of 2.5% was somewhat arbitrarily chosen so give some room for fluctuations, while still being restrictive enough to avoid too many false negatives.

<table>
<thead>
<tr>
<th>Use-case</th>
<th>Control 1, Violation Rate</th>
<th>Control 2, Violation Rate</th>
<th>Control 3, Violation Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Home</td>
<td>0.0%</td>
<td>0.1%</td>
<td>0.3%</td>
</tr>
<tr>
<td>Vets</td>
<td>0.6%</td>
<td>0.8%</td>
<td>0.9%</td>
</tr>
<tr>
<td>Find</td>
<td>0.5%</td>
<td>0.4%</td>
<td>0.4%</td>
</tr>
<tr>
<td>View Owner</td>
<td>0.4%</td>
<td>0.2%</td>
<td>0.5%</td>
</tr>
<tr>
<td>Add Pet</td>
<td>0.6%</td>
<td>0.6%</td>
<td>0.4%</td>
</tr>
<tr>
<td>Add Visit</td>
<td>0.8%</td>
<td>0.7%</td>
<td>1.1%</td>
</tr>
<tr>
<td>Edit Owner</td>
<td>1.1%</td>
<td>1.2%</td>
<td>1.8%</td>
</tr>
<tr>
<td>Add Owner</td>
<td>0.4%</td>
<td>0.6%</td>
<td>0.4%</td>
</tr>
</tbody>
</table>

Table 6. Violation rates for control load tests.

For the load tests simulating various performance issues, the control chart plugin was configured to use the baseline and violation rate as outlined above. In the next chapters, the results for each such performance problem is described.

**Excessive Dynamic Allocation**

“Table 7” below shows the violation rates for the three test runs against the version of the sample application with the excessive dynamic allocation bottleneck. Test cases where the violation rate exceeds the maximum allowed value of 2.5% are highlighted in bold font.
Since database connections are initialized as part of all use-cases except “Home”, it is expected to see a large number of cases where the test’s acceptance criterion is not met. Control charts for the control and excessive dynamic allocation runs of the “Edit Owner” use-case are shown in “Figure 9” and “Figure 10” below. By comparing these, the increase in response times is clearly visible. Note that some outlier data points have been cutoff to make the charts more readable.

<table>
<thead>
<tr>
<th>Use-case</th>
<th>Run 1, Violation Rate</th>
<th>Run 2, Violation Rate</th>
<th>Run 3, Violation Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Home</td>
<td>1.4%</td>
<td>1.5%</td>
<td>1.4%</td>
</tr>
<tr>
<td>Vets</td>
<td>41.4%</td>
<td>37.7%</td>
<td>35.0%</td>
</tr>
<tr>
<td>Find</td>
<td>4.1%</td>
<td>2.7%</td>
<td>2.5%</td>
</tr>
<tr>
<td>View Owner</td>
<td>6.7%</td>
<td>6.3%</td>
<td>5.8%</td>
</tr>
<tr>
<td>Add Pet</td>
<td>77.8%</td>
<td>75.6%</td>
<td>72.1%</td>
</tr>
<tr>
<td>Add Visit</td>
<td>43.7%</td>
<td>43.3%</td>
<td>40.2%</td>
</tr>
<tr>
<td>Edit Owner</td>
<td>16.5%</td>
<td>13.8%</td>
<td>13.4%</td>
</tr>
<tr>
<td>Add Owner</td>
<td>3.8%</td>
<td>5.1%</td>
<td>2.4%</td>
</tr>
</tbody>
</table>

*Table 7. Excessive dynamic allocation results.*

*Figure 9. Control chart for the control run of the "Edit Owner" use-case.*
Figure 10. Control chart for the excessive dynamic allocation run of the "Edit Owner" use-case.

N+1 Database Selects

“Table 8” below shows the violation rates for the three test runs against the version of the sample application with the n+1 database select bottleneck. Test cases where the violation rate exceeds the maximum allowed value are highlighted in bold font.

<table>
<thead>
<tr>
<th>Use-case</th>
<th>Run 1, Violation Rate</th>
<th>Run 2, Violation Rate</th>
<th>Run 3, Violation Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Home</td>
<td>0.0%</td>
<td>0.1%</td>
<td>0.1%</td>
</tr>
<tr>
<td>Vets</td>
<td>0.5%</td>
<td>0.4%</td>
<td>0.9%</td>
</tr>
<tr>
<td>Find</td>
<td>24.1%</td>
<td>18.8%</td>
<td>21.1%</td>
</tr>
<tr>
<td>View Owner</td>
<td>0.5%</td>
<td>0.3%</td>
<td>0.5%</td>
</tr>
<tr>
<td>Add Pet</td>
<td>1.2%</td>
<td>1.5%</td>
<td>2.3%</td>
</tr>
<tr>
<td>Add Visit</td>
<td>1.2%</td>
<td>0.9%</td>
<td>1.1%</td>
</tr>
<tr>
<td>Edit Owner</td>
<td>1.7%</td>
<td>1.0%</td>
<td>1.4%</td>
</tr>
<tr>
<td>Add Owner</td>
<td>0.3%</td>
<td>0.4%</td>
<td>0.9%</td>
</tr>
</tbody>
</table>

Table 8. N+1 database selects results.

The n+1 bottleneck was implemented for the “Find” use-case, which is the only scenario where the maximum violation rate is exceeded. This shows that the chosen method is able to identify performance issues at a granular level and thus in certain cases can be used to aid an engineer in pinpointing the performance issue. However, some of the other use-cases show higher violation rates compared to the control suite, which indicate that there also may be risk of false positives on a per use-case level.

Control charts for the control and n+1 database select runs of the “Find” use-case are shown in “Figure 11” and “Figure 12” below. Note that some outlier data points have been cutoff to make the charts more readable.
Figure 11. Control chart for the control run of the "Find" user-case.

Figure 12. Control chart for the n + 1 database selects run of the "Find" use-case.

**Serialized Access**

“Table 9” below shows the violation rates for the three test runs against the version of the sample application with the serialized access bottleneck. Test cases where the violation rate exceeds the maximum allowed value are highlighted in bold font.

<table>
<thead>
<tr>
<th>Use-case</th>
<th>Run 1, Violation Rate</th>
<th>Run 2, Violation Rate</th>
<th>Run 3, Violation Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Home</td>
<td>0.2%</td>
<td>0.2%</td>
<td>0.2%</td>
</tr>
<tr>
<td>Vets</td>
<td>0.9%</td>
<td>1.5%</td>
<td>1.0%</td>
</tr>
<tr>
<td>Find</td>
<td>0.2%</td>
<td>1.4%</td>
<td>0.9%</td>
</tr>
<tr>
<td>View Owner</td>
<td>4.2%</td>
<td>5.4%</td>
<td>4.4%</td>
</tr>
<tr>
<td>Add Pet</td>
<td>2.0%</td>
<td>2.0%</td>
<td>1.4%</td>
</tr>
<tr>
<td>Add Visit</td>
<td>1.4%</td>
<td>1.4%</td>
<td>1.0%</td>
</tr>
<tr>
<td>Edit Owner</td>
<td>2.3%</td>
<td>1.6%</td>
<td>1.3%</td>
</tr>
<tr>
<td>Add Owner</td>
<td>0.9%</td>
<td>1.4%</td>
<td>1.1%</td>
</tr>
</tbody>
</table>

*Table 9.Serialized access results.*
The serialized access bottleneck was implemented for the “View Owner” use-case, which is the only scenario where the maximum violation rate is exceeded. Control charts for the control and serialized access runs of the “View Owner” use-case are shown in “Figure 13” and “Figure 14” below. Note that some outlier data points have been cutoff to make the charts more readable.

![Figure 13. Control chart for the control run of the "View Owner" user-case.](image1)

![Figure 14. Control chart for the serialized access run of the "View Owner" use-case.](image2)

**Misconfigured Cache**

“Table 10” below shows the violation rates for the three test runs against the version of the sample application with the misconfigured cache bottleneck. Test cases where the violation rate exceeds the maximum allowed value are highlighted in bold font.
<table>
<thead>
<tr>
<th>Use-case</th>
<th>Run 1, Violation Rate</th>
<th>Run 2, Violation Rate</th>
<th>Run 3, Violation Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Home</td>
<td>0.1%</td>
<td>0.0%</td>
<td>0.2%</td>
</tr>
<tr>
<td>Vets</td>
<td>90.1%</td>
<td>88.6%</td>
<td>89.1%</td>
</tr>
<tr>
<td>Find</td>
<td>0.2%</td>
<td>0.3%</td>
<td>0.2%</td>
</tr>
<tr>
<td>View Owner</td>
<td>0.4%</td>
<td>0.3%</td>
<td>0.6%</td>
</tr>
<tr>
<td>Add Pet</td>
<td>0.7%</td>
<td>0.7%</td>
<td>1.3%</td>
</tr>
<tr>
<td>Add Visit</td>
<td>0.9%</td>
<td>0.6%</td>
<td>0.8%</td>
</tr>
<tr>
<td>Edit Owner</td>
<td>1.6%</td>
<td>1.1%</td>
<td>1.4%</td>
</tr>
<tr>
<td>Add Owner</td>
<td>0.7%</td>
<td>0.1%</td>
<td>0.7%</td>
</tr>
</tbody>
</table>

*Table 10. Misconfigured cache results.*

The caching logic was removed from the “Vets” use-case, which is the only scenario where the violation threshold is exceeded.

Control charts for the control and misconfigured cache runs of the “Vets” use-case are shown in “Figure 15” and ”Figure 16” below. Note that some outlier data points have been cutoff to make the charts more readable.

*Figure 15. Control chart for the control run of the "Vets" user-case.*
Missing Database Index

“Table 11” below shows the violation rates for the three test runs against the version of the sample application with the missing database index bottleneck.

<table>
<thead>
<tr>
<th>Use-case</th>
<th>Run 1, Violation Rate</th>
<th>Run 2, Violation Rate</th>
<th>Run 3, Violation Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Home</td>
<td>0.2%</td>
<td>0.2%</td>
<td>0.1%</td>
</tr>
<tr>
<td>Vets</td>
<td>1.0%</td>
<td>0.7%</td>
<td>1.0%</td>
</tr>
<tr>
<td>Find</td>
<td>0.9%</td>
<td>1.1%</td>
<td>1.1%</td>
</tr>
<tr>
<td>View Owner</td>
<td>0.7%</td>
<td>0.5%</td>
<td>0.9%</td>
</tr>
<tr>
<td>Add Pet</td>
<td>0.8%</td>
<td>1.1%</td>
<td>1.4%</td>
</tr>
<tr>
<td>Add Visit</td>
<td>0.9%</td>
<td>1.1%</td>
<td>1.3%</td>
</tr>
<tr>
<td>Edit Owner</td>
<td>1.8%</td>
<td>1.3%</td>
<td>0.8%</td>
</tr>
<tr>
<td>Add Owner</td>
<td>0.7%</td>
<td>0.4%</td>
<td>0.9%</td>
</tr>
</tbody>
</table>

Table 11. Missing database index results.

The index that was removed from the database affected the “Find” use-case and thus it was expected to find an increased violation rate for that scenario. However, even though there was a slight increase compared to the control runs, it was not enough to exceed the violation threshold and fail the load test.

After some further investigation, it was detected that the original implementation of the “Find” use-case used a non-optimal strategy for fetching owners and their corresponding pets and visits. This performance issue was clearly visible in long-running load tests, but not when running for only six minutes, as was the case here.

Additionally, the implementation used a strategy for searching for owners that did not utilize the database index to its full extent. Consequently, removing the index had a very small effect on the overall performance.

To validate this theory, the bottleneck in the original implementation was corrected and the database query was modified to fully utilize the index. Based on this, a new baseline was established. Removing the index and repeating the load test after this change did
indeed produce a slightly increased violation rate, but still not enough to exceed the 2.5% threshold.

Some further experimentation with the size of the database data set indicated that the initial data set was too small for the index to have a major impact on performance. After increasing the data set size by roughly a factor of 10, establishing a new baseline and re-running the test, the violation rate for the “Find” use case increased to 5%, i.e. above the defined threshold.

**Summary of Results and Conclusion**

The chosen method was able to flag performance issues in at least four out of the five simulated scenarios. Due to a faulty setup of the missing database index scenario, the results from that experiment was not conclusive. However, even this scenario supports the chosen approach, at least to some extent, since no false positives were reported.

Going back to the requirements listed in the “Problem Statement” chapter, it is now possible to evaluate the results against the stipulated requirements:

1. *Fully automated.* All the steps in the implemented continuous delivery deployment pipeline were fully automated. The only manual step was to configure the control chart plugin with the baseline metrics as part of the pipeline setup.
2. *Fast feedback.* For the chosen sample application, load test feedback was given within approximately six minutes after triggering the test run. This should be considered acceptable, especially in contrast to traditional load testing where it, according to the author’s experience, often takes hours to setup and execute a single test run.
3. *High level of transparency.* As previously stated, control charts illustrate the results of a load test in a way that is easily understood.
4. *Tunable sensitivity.* By setting a violation rate threshold when configuring the control chart plugin, it is possible to tune the sensitivity of the algorithm.
5. *Reliable.* In at least four out of the five simulated scenarios, the chosen method was able to correctly detect the introduced performance issue. The results were also consistent; for all test suites, either all three tests were flagged as failures or all three tests were flagged as successful.

Therefore, the conclusion from this degree project is that using a method based on control charts is a viable approach to automated verification of load test results in a continuous delivery deployment pipeline.
Discussion

The results from the previous section show that the chosen control chart approach is able to detect introduced performance issues in several cases. Being able to do this within six minutes should also be considered an acceptable timeframe for a continuous delivery pipeline.

However, it should be noted that the sample application used for this work is considerably smaller and less complex compared to many real-world applications. It is not unlikely that applying this approach to a more complex system would be more challenging and for example require longer test runs or produce less consistent results. Nevertheless, in the author’s opinion, there is nothing inherent to the chosen method that prevents it from being used successfully in a more complex scenario as well.

A related area of research that could be investigated further is whether it provides value to include other types of performance metrics, such as server CPU utilization, memory consumption and disk I/O as well. Response time metrics has the advantage of being readily available from most load testing tools. In contrast, many of the other types of metrics would need specialized data gathering tools and frameworks.

A key feature of the load tests considered in this thesis is to quickly be able to provide feedback regarding introduced bottlenecks. A lesson learned from the missing database index experiment is that this type of rapid load testing needs to be supplemented by other types of load tests, for example stress and soak tests, in order to cover the full spectrum of possible performance issues. Also, in many instances there may be a tradeoff between having a fast feedback cycle, and thus shorter test duration, versus catching a wider variety of performance bugs by having longer tests or put more stress on the system (which may introduce more variance to the results).

A related challenge is the one of finding a good violation rate threshold. Too high, and performance issue may be missed, while having a too low threshold increases the probability of false alarms. In both cases, there is a major risk of people not trusting or even ignoring the results as well as increased test maintenance.

To make informed decisions in these areas, it is the author’s opinion that good domain knowledge and a holistic understanding of the entire system are still required to produce desirable results. Additionally, to find an optimal test setup, a fair amount of experimentation and tuning are likely key ingredients as well.
References


