Bachelor Thesis

September 27, 2017

Continuous Web Performance Engineering

An industrial case study on end-user load testing

Lukas Bösch

of Wangen, Schweiz (10-719-557)

supervised by

Prof. Dr. Harald C. Gall

Christoph Laaber

University of Zurich[UZH]

s.e.a.l.

software evolution & architecture lab

Bachelor Thesis

Continuous Web Performance Engineering

An industrial case study on end-user load testing

Lukas Bösch

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Bachelor Thesis

Author: Lukas Bösch, lukas.boesch90@gmail.com

Project period: 01.04.2017 - 01.10.2017

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

It has been a long time since the start of this thesis. During the time period working on it I have learned a lot about scientific research and performance engineering. However without the involvement of many other people I would not be where I am right now.

At first I want to thank my colleagues from the performance engineering team in the financial institution. They gave me a lot of valuable input in the field of performance engineering and load testing. Especially Frank and Josef supported me during the creation of this thesis with their practical experience in this field. I also got valuable linguistical feedback from Johnny and Svenja. Many thanks go to my supervisors from the university and the company.

Last but not least I want to thank my supervisor Christoph Laaber who gave me valuable input to the scientific research of performance engineering. He also helped me creating this document with his academic experience.

Thanks a lot to all of you!

Abstract

The evolution of software development methodologies affects all parties involved. Shifting from long-iterative, big bang models to continuous, agile methodologies leads to different challenges, advantages and disadvantages. This thesis focuses on performance engineering in a continuous development methodology. Therefore a case study is conducted in a financial institution where this change is ongoing. The process of end-user load testing on the browser API is presented. It is analysed in what extent it is possible to integrate it in the continuous methodology. Opportunities to automate different steps are explicitly looked out for in order to further increase the efficiency of the load testing process as the main challenge faced is the closer time restriction. At the end a quantitative evaluation is done based on the conducted case study. The previous approach will be compared to the presented approach. The evaluation of the case study shows that the presented approach costs less than the previous approach.

Zusammenfassung

Eine Veränderung des Softwareentwicklungsprozesses hat Einfluss auf alle involvierten Parteien. Der Wechsel von lang-iterativen, 'big bang' Modellen zu kontinuierlichen, agilen Methoden bringt verschiedene Herausforderungen sowie Vor- und Nachteile mit sich. Diese Bachelorarbeit betrachtet die Sicht eines Performance Engineers als Teil innerhalb dieses Prozesswechsels. Veranschaulicht wird diese Sicht in einer Fallstudie bei einer finanziellen Institution durchgeführt. Der bisher verwendete Prozess des Lasttestens wird im Bezug auf die Aufgabenstellung präsentiert. Des weiteren wird analysiert, in welchem Ausmass es möglich ist, diesen Prozess in eine kontinuierliche Methodik zu integrieren. Möglichkeiten zur Automatisierung einzelner Schritte dieses Prozesses werden aufgezeigt, um der engeren Zeitrestriktion gerecht zu werden. Am Schluss wird eine quantitative Evaluation, welche auf der Fallstudie basiert, durchgeführt. Der bisherige Ansatz wird mit dem präsentierten verglichen mit dem Ergebnis, dass der präsentierte Ansatz in der Fallstudie besser ist als der bisherige.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

The motivation for this thesis is based largely on the widespread adaption of agile software development methodologies across all industries. The shift from big bang approaches like the waterfall model to faster iterative models like Scrum is challenging all involved parties from the developers to the testers and finally to operations [46]. Also there is a shift in project organization as the developer, tester and operation responsible is not strictly separated anymore. Continuous integration requires a fast flow of information between the parties hence moving closer to each other to minimize communication overhead.

This shift also influences the whole testing process. There is the need to test and deliver results as fast as possible to reduce the time-to-market. The shift to an iterative approach makes the interpretation of test results of software functionality fairly straight-forward, yet the design and analysis of performance testing poses new challenges. Load testing requires more resources to execute as there is the need to have a dedicated environment to deliver significant and accurate results. The test design decision can differ for each individual application and is highly dependent on the non-functional requirements of the application. In order to verify the non-functional requirements, the test implementation is needed on the browser's GUI level to measure true end-user response times including the browser processing time.

The automation of load testing activities includes test implementation, execution and analysis. This thesis tries to deliver possible solutions to integrate performance tests in an agile, continuous web development process. The focus for the test implementation is laid on client-based scripting on the GUI level in this thesis. To evaluate the effectiveness of the solutions provided a case study is conducted in one project. This happens in a financial institution where over 70 % of the performance tests are executed through the browser API and therefore implemented on the GUI level.

1.1 Motivation

Motivating Example. John Doe is a performance engineer in a financial institution. He receives a load test request from an application inside the financial institution. The developing team of the application uses the waterfall model. They are engineering for two and a half months and want to assure their software's quality. This should happen in the last two weeks before they plan to go live during the user acceptance test stage. John receives the use case and respective test data and immediately starts planning and implementing his load test. After he implemented the end-user use case on the GUI two days have already passed. He defines a test plan according to the time he is given for testing. He plans to execute three different test

runs with different workload definitions and execution duration. So from the third to the fifth day he is executing his planned test runs and analyzes the captured metrics. John detects that the throughput was not reached in all of the test runs and the login response time was longer than expected. Through further analysis of these incidents he discovers an implementation error leading to the abortion of several use case executions. In the server side monitoring he also identifies a long running query on the database. He reports this incident to the developing team. On the eighth day, three days before the release, he can retest as the incidents were reported as solved. The retest and the analysis are finished on the tenth day and last day of testing. The beforehand identified incidents were retested successfully. However through reaching the expected throughput for the test runs, John further identifies a deadlock exception during one test execution. But there is no more time for fixing or retesting this incident according to the release plan. So the developing team now has two possibilities. They can delay the go live to fix and retest the occurred deadlock exception. The second option is to accept that these deadlock exceptions will occur and go live according to the release plan. Either of these options is optimal.

Now imagine the developing team would shift to the SCRUM framework [56] instead of the waterfall model. What changes would apply to John Doe? Could he still implement his load tests in the same way? These two questions are put together into research questions 1 in 1.2. Further what challenges does he face? Are there any advantages or disadvantages compared to the load testing with the previous approach? These questions are reflected in research question 2 in 1.2.

Motivating Research. In web applications in general the psychological aspect is an important driver in optimizing performance. According to Nielsen [47] "The numbers about human perception and response times have been consistent for more than 45 years." He uses the term response time which equals the time a user waits after an action until he can interact with the software again. According to his research the illusion of an instant response time is below 0.1 seconds. A response time between 0.1 and 1 second load times keep our thoughts flawless. At response times from 1 to 10 seconds we can barely keep our attention. Any further increase of response time leads to complete loss of our attention. This aspect is especially critical in daily used business applications where employees need to keep their focus on the job at hand which is nearly impossible with page load times that are higher than 1 second or even 10 seconds. This is not only influencing their current speed of work but also their long-term happiness and mental fatigue with the workplace and firm.

One could argue that during long load times you can perform other tasks. Czerwinski [18] in her study on workplace interruptions refutes that this is really helping productivity as the switching of tasks is psychologically exhausting and for every iteration the time to resume the original task increases disproportionately also decreasing motivation and developing mental fatigue. Another study was performed by Züger et al. [79] where they evaluated the disruptiveness of knowledge workers.

The behaviour of our mind is not the only aspect that influences the perception of response times. The rising expectations of users in the past decade also indicates that more focus needs to be laid on software performance. This is shown through two different studies Akamai [4] from 2006 stated that the average online shopper expected pages to load in 4 seconds or less. The second and more recent one from 2014 [26] states that 49 % of users expect pages to load in 2 seconds or less.

Slow Performance, which includes load times and response times, does not only have an impact on users perception and mental health it can also lead to a permanent abandonment of the Website. As Akamai [3] states slow performance even leads to a higher number of permanent

abandonments than temporary outages.

Performance not only impacts the user's behaviour and perception but also impacts applications which gain revenue on sites. Every 100ms of latency on amazon.com costs them 1% sales. The duration to load google results also result in a drop of traffic: "Half a second delay caused a 20% drop in traffic. Half a second delay killed user satisfaction." [40]. Google also correlated latency with the number of searches per user which decreases by 0.2% to 0.6% with a web search latency of 100 to 400ms. [12] Performance can even impact the download rate of a browser as Mozilla showed in their statistic. [59].

1.2 Research Questions

Driven by the previously elicited problem statements, this thesis covers two research questions:

Research Question 1: To what extent is it possible to integrate the process of performance testing focusing on browser Application Programming Interface (API) test implementation in a development process which follows the SCRUM framework?

This question covers an explanation of the current process of load and performance testing. It asks what changes are necessary to apply when integrating it in an early stage of the development process. The process will be integrated in a project inside the financial institution and shown as a case study. A focus in answering this question will be laid on the analysis of the load tests including the path to identify the root cause of performance incidents.

Research Question 2: What are the challenges, advantages and disadvantages of an earlier integration of load and performance testing?

A general analysis of this approach and a comparison to the current load testing process will be performed in order to answer this question. The main challenge is the time constraint as the time window to execute and analyse load and performance tests is dependent on the release cycles. Therefore the process needs to be optimized time- and effort-wise. Opportunities in automating single steps of the load testing process will be identified and presented to conquer the time constraint for the continuous testing approach.

1.3 Scope

The emphasis in this thesis is laid on the load and performance test process with the browser API test implementation. The reason of the focus on browser API test implementation lies in the completeness of the test. It is able to measure the true end-to-end performance which will be observed by the user including JavaScript executions on the DOM-tree and rendering times. It also enables the complete analysis of measured slow response times from the client. This

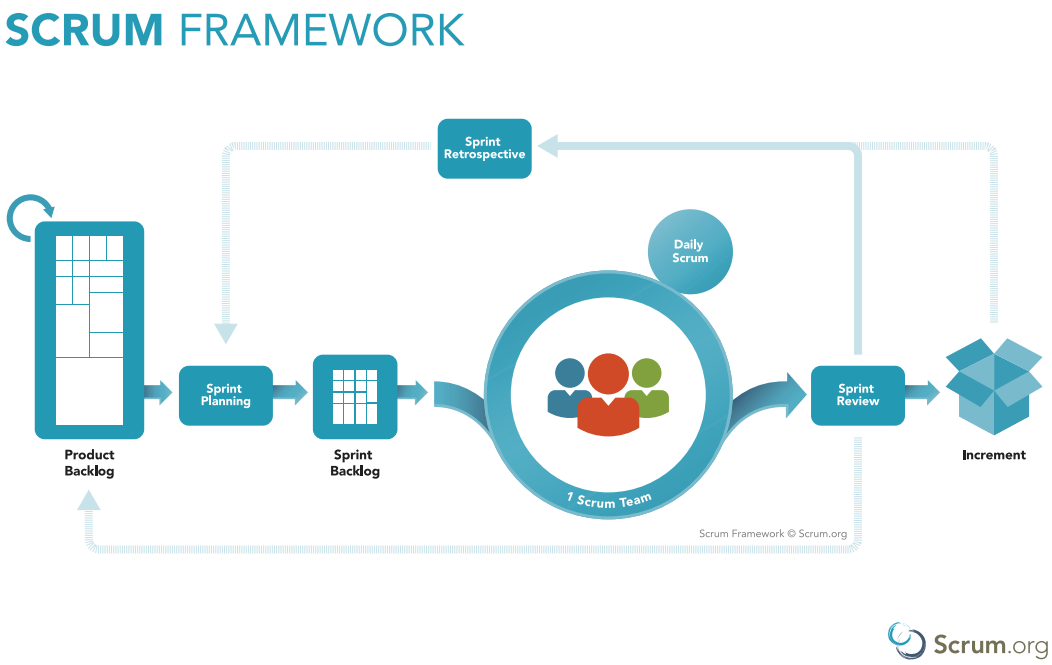

Figure 1.1: SCRUM Sprint Overview
[56]

includes the measurement of true asynchronous requests which is difficult to implement with a protocol based test over HTTP.

The thesis covers the whole process including the prerequisites, design, client-side metrics, server-side metrics and the analysis of a browser API load and performance test. The main idea is to implement it in the agile development framework SCRUM where potentially shippable product increments are deployed after each sprint as depicted in figure 1.1 At the end of a sprint the potentially shippable product is not necessarily released. It is an increment that is fully functional making it ready for browser API performance tests.

During the description of the process suggestions for optimizing the single steps are presented to further decrease the overall effort. This will also be further shown and evaluated through a case study. In the case study a browser API load test is implemented in a project within a financial company. This project follows the SCRUM framework with potentially shippable product increments created every two weeks.

1.4 Background of the Financial Institution

The case study is taken in a financial institution which both develops their own software for internal usage, as well as purchasing packaged software. The author of this thesis is employed as a part time employee during the period of time where the case study was taken. This section gives a quick overview of the financial institution. The financial institution is globally present

with the headquarter located in Zurich. Amongst others, offices are located in Frankfurt, Hong Kong, London, Singapore, Tokyo and Montevideo. The number of employees rapidly increased during the last decade. Until the creation of this thesis the company employs over 6000 people. 70 % of the in-house applications are using a browser interface which is an additional driver of this thesis to focus on web performance testing.

1.5 Structure of the Thesis

This thesis will be structured in six chapters:

1. Introduction: The introduction contains the main motivations and describes the research questions.
2. Related Work: This chapter gives an overview of the recently performed research in the field of performance engineering. It covers the whole process of load testing, from test implementation to the analysis, different challenges in web performance and research on the usage of Application Performance Management (APM) tools.
3. Approach in this Thesis: This thesis' general approach is shown. An in-depth view of the requirements, test implementation, test planning, test execution and the analysis will be given.
4. Case Study: The approach will be integrated in a project at the company. This chapter shows the complete implementation and results of the implemented tests will be presented focusing on the analysis and issue identification.
5. Evaluation: An evaluation of this approach based on the case study provided will be taken.
6. Closing Remarks: In the final chapter a conclusion of this approach will be drawn and future research questions posed.

Chapter 2

Current State-of-the-Art Performance Engineering

In this chapter I will provide the current state-of-the-art performance engineering activities of the scientific community, the industrial standards and the current standard process in the company. The focus in these evaluations is laid towards the process and its single steps of the load and performance testing. Therefore excluding scientific research of browser API testing as this is specific for the conducted case study. At the end, a comparison of the scientific research and the industrial processes is done based on the expositions made in the first two sections.

2.1 Scientific Research

This section delivers an overview of the scientific research that is related to this thesis. The findings are then used to compare the scientific research with the industrial best-practices. The scientific research is further presented in chapter 3.

Merriam-Webster Online Dictionary delivers a definition of the term science [19]: "knowledge or a system of knowledge covering general truths or the operation of general laws especially as obtained and tested through scientific method". As per this definition, science is a "state of knowing", gaining knowledge through observation and verify the gained knowledge with scientific methods. In the scientific research of performance engineering and information technology in general there additionally is the delivery of possible solutions to these observations. Further examples of scientific research is given in the continuation of this section.

Research is made on creating performance models to try to predict a system's performance based on architectural design [65] or load test results [37] [32].

Effort is also spent on identifying typical performance anti-patterns and their root cause [50] [73] [65]. They deliver valid solutions in identifying these common anti-patterns and also help in finding the root cause of these issues. These common anti-patterns include Blob, Circuitous Treasure Hunt, Empty Semi Trucks, Tower of Babel, One-Lane Bridge, Excessive Dynamic Allocation, Traffic Jam, The Ramp and More is Less.

Oftentimes, scientific research focuses on identifying isolated performance problems and their root cause but there is also research in the field of automatic analysis of load test results and the automatic modification of load test results [32]. Driven by industrial requirements there is also research on current APM tools and the automatic identification of performance problems according to the data collected by APM tools such as AppDynamics [6] or DynaTrace [21] [5] [35] [1].

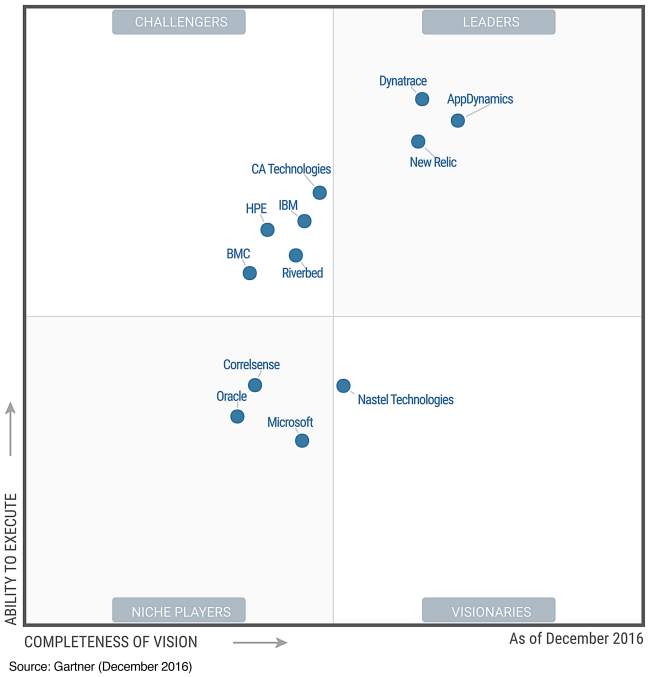

Figure 2.1: Magic Quadrant for APM suites 2016
[28]

Research on GUI-level testing strategies is also done by the scientific community either on automating GUI test scripts [52] or by bringing the load test design closer to real usage by examining real user thinktimes [53].

2.2 Industrial Best-Practices

The focus in this section will be laid towards the industrial best practices for load and performance testing.

To identify the current industry standards regarding tooling support we can take a look on the currently successful vendors of standard performance testing software and APM. Assuming that the demands of the market meet the supplies of the vendors according to the supply and demand principle. In order to achieve that we take a look on the Gartner Magic Quadrant [27] of 2016 about APM which is depicted in figure 2.1.

The fact that there exists an own magic quadrant for APM tools shows the importance of performance monitoring in the industry. This includes the monitoring of productive environments as well as the analysis of load testing activities during pre-production phases. This versatility in usage and the unused potential are other reasons why APM tools relish increasing attention in the industry and define a performance monitoring standard.

As software in general is a supportive service for the industry there exists the concept of service-level agreement (SLA) [58] to define quality metrics that the software needs to fulfill. In order to validate SLA's before deploying a software, load and performance testing can be

performed.

The goal is not laid towards identifying certain performance problems. Companies want to assure that the application can cope with the load it has to deal with. In order to achieve this it is important that the performance test come as close to the productive usage of an application as possible by implementing real end-user use cases on the browser interface as tests. As reliably as possible one is able to make a statement and a prediction on how the application will perform including back-end and front-end performance, especially including rendering times and asynchronous requests.

2.3 Comparison of the current Industry Best-Practices and the Scientific Research

As the industry is practice focused on the utilization of applications and science is more general (or theoretical), Merriam-Webster delivers another definition of science linking it with practice [19]: "a system or method reconciling practical ends with scientific laws". This definition states that science can also be defined as the formal explanation of practical ends. For performance engineering, scientific research is able to observe the practical use, analyse and formalize it and then deliver possible solutions.

In other words, research is able to deliver results on different requirements that the industry poses acting as a problem solving instance. An example for this can be found on the research spent in APMs as there is scientific effort spent in the generalisation of the execution trace which is done by different APMs. If there is a general representation of execution stack traces it will be possible to automate the analysis of them [48].

For further comparison the scientific research presented in this thesis is compared to the solutions provided by an APM provider. The scientific research oftentimes delivers solutions to single problems. The APM solution is able to gather monitoring data of one application out of the box. Therefore it is possible to further identify performance problems based on the data collected. Since this data is collected within one data center for different tiers the automatic analysis can even include monitoring data from other tiers. On the other hand scientific research does not deliver this kind of overview and consideration of multi-tier analysis. For large multi-tier systems an APM is a very useful tool to identify performance incidents and help in analyzing their root cause fast. However for optimizing single tiers, like a database, scientific, open-source solutions are the better option. Also APM solutions are very expensive and therefore present a good tool for companies who can afford them. Another pitfall of APM solutions are the non-trivial usage as it offers such a vast field of functionality and data. Therefore it is hard for a developer to effectively use an APM to find what he desires without any instructions.

As a summary APM is able to give an overview of large software systems. When the limits are reached and the incident is categorized, scientific research helps in optimizing incidents or single tiers. APMs are not yet ready to be used effectively by a developer whereas scientific research delivers solutions which can be used directly by integrating performance metrics in the development environment. An example can be found by Cito et al. [15] where they made performance metrics accessible through an eclipse [25] plugin. In practice there can occur simple slips which also need to be identified. For example if the log level 'debug' is deployed the CPU usage is heavily increased for each transaction. These kind of problems are rarely dealt with in scientific research but are problems faced in practice which need to be identified and solved.

2.4 Performance Testing in the Financial Institution

In this financial company there are already several testing activities integrated in the project team. Performance testing is not part of the development process in project teams yet and is performed by a separate team independent of the project team. The performance testing is always the last stage where it is already very hard to fix some performance issues. Through the presented approach we want to deliver a solution to take a step to the desired state where performance testing activities are included in the development process. This has several key advantages. The developing team will be able to detect performance issues closer to the point of time where it was introduced. The developer is then able to better remember his own thoughts during the development of the code which caused the performance incident in the first place. Assuming that the performance incident is due to an erroneous design decision the design can be rethought at a point in time where it is quite easy to redesign the software. Zimran [78] states:"The earlier the detection of performance problems or dependencies the less expensive they are to fix.". Amongst others this can include design patterns chosen or database table design decisions. Especially database table design can influence performance significantly if the size and growth of the tables during the execution times are not considered. If such an issue is detected close to the point of development the redesign can happen with much less refactoring work than if it was discovered when the functionality of the application is completely developed.

Functional end-user test cases and unit tests are common practice but separated from performance testing activities. Although test cases that could be used for performance testing are already available, dedicated use cases for performance tests are developed separately and therefore this test development work is duplicated. This is historically grown as performance testing started out as fire fighting activities after end-user complaints during production. By now performance testing already shifted left in the software life-cycle (from after deployment to pre-deployment). This approach wants to further shift-left and enable performance testing in a developing phase. As the progression is by now with the consideration of continuous deployment and development in the future these phases will move closer to each other and most probably to a point where they can't strictly be distinguished. An overview of the current testactivities in the development phase is shown in figure 2.2. It shows the amount of test activities conducted in each phase of the software life-cycle. Additionally, the different testing environments are shown in the boxes below the life-cycle phases. It is visible that the current performance test activities are the highest during the test phase of the realization. Especially performance test activities are started at the UAT environment during the test phase.

Example of the Previous Approach. In non-agile developing teams the company standard release plan contains four releases per year. An exemplary release plan is found in figure 2.3. During these three month cycles of development the performance testing phase has a duration of two weeks in the UAT phase. This includes the test design, implementation, execution and analysis. As we can already see there is more time in the system test phase where no performance testing is happening as of now. As depicted in section 1.3 this thesis tries to apply to an agile development process and utilize the potentially shippable product increment for testing every two weeks. The different phases in this exemplary release plan happen in every cycle depicted in figure 1.1. The test environments are built up and maintained throughout the development process making such a release plan obsolete. This fact also enables the continuous testing throughout the development process. In other words this shift-left of testing activities is

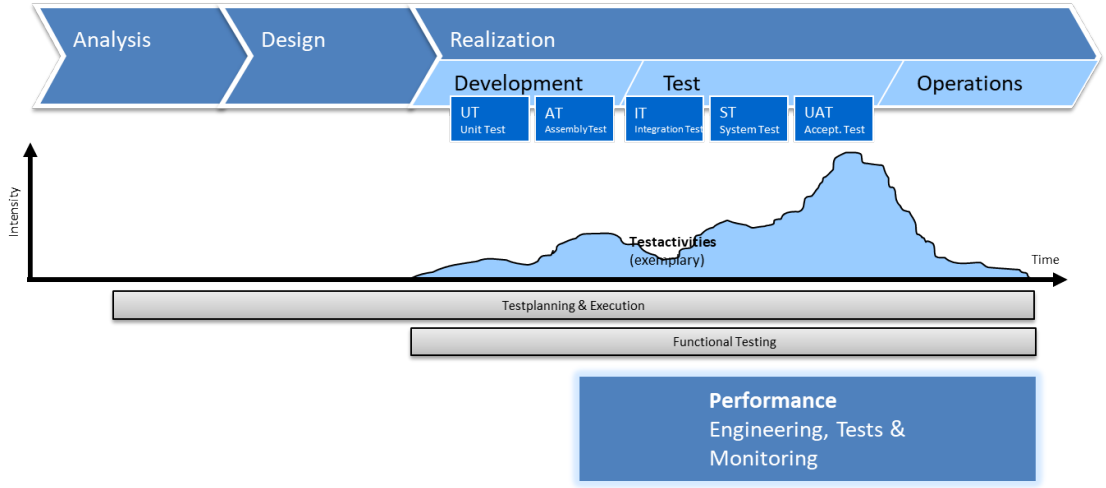

Figure 2.2: Current Testactivities in the Development Process

enabled by the change of the development process. This fact theoretically answers the research question 1 in section 1.2. The practical evaluation in the case study is done in subsection 6.1.2.

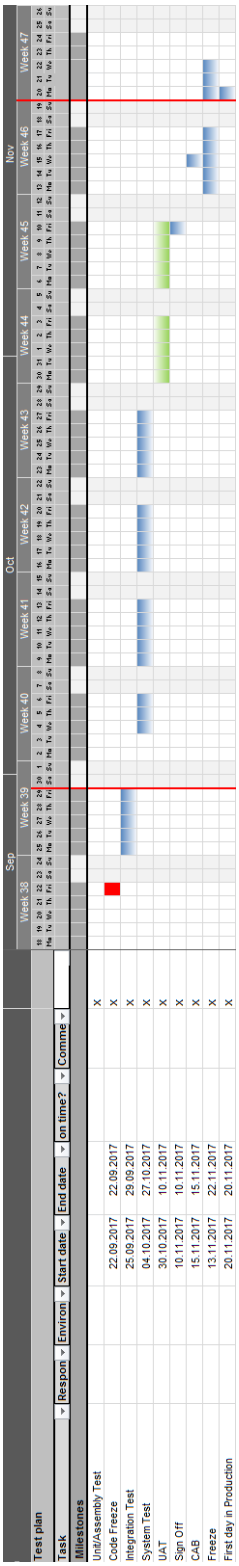

Figure 2.3: Exemplary Release Plan of a Trimestrial Release

Chapter 3

Related Work about Performance Engineering

This chapter provides a brief overview of current scientific research topics, should give an overview on the challenges faced for this thesis and present additions to the approach taken in this thesis. Starting with the APM because it is mentioned and referred to in the subsequent sections. The subsequent sections cover the different steps of the performance test process: Implementation, performance evaluation, load test. Finally different current challenges in web performance are discussed as this the thesis focuses on web performance load testing.

3.1 Application Performance Management

This section covers scientific related work about the use of APMs. Since this approach uses an APM solution for the server-side analysis they are highly relevant for this thesis. For large-scale-systems and web applications APM Tools are important tools to ensure software quality and monitor the application during the operations phase. Of course they can also be used to analyse load test results prior to a release as a load test just tries to simulate the productive phase of a system. Therefore oftentimes the same methods as research presents for the analysis of APM data during operations phase can be used.

The effectiveness of APM Tools (AppDynamics [6], New Relic [54], Dynatrace [21] and Kieker [39]) is evaluated by Ahmed et al. [1] with sample java applications. The test workload stayed the same but the code between the benchmarks was modified with performance regressions. They evaluated which APM Tool is able to detect and find the root cause of the previously inserted regression. The APM Tools were able to detect the performance regressions but identifying the root cause was challenging and time consuming. Especially when mining a large data set they identified the lack of common APM tools of manually extending their functionality individually. In our case of load testing business applications with a limited number of users this is only a minor issue in both testing/development phase and production phase.

Since there are a lot of stakeholders involved in a software development process especially in large companies there has been research on how APM data can be made more accessible to people with little to no technical background or knowledge. Heger et al. [34] state: "APM is not a purely technical topic anymore, as there is also a need for support of business activities and vice versa". Walter et al. [71] present research in this direction. They have the vision of declarative performance engineering, where the user can address his concerns with a high level language, which then automatically translates to the respective methods, techniques and

tools of the established performance engineering. This requires a high grade of automation of performance engineering activities which is a similar goal that this thesis tries to achieve.

Research focusing on the analysis of APM data is performed by Angerstein et al. [5]. They focus on large enterprise systems and a very high amount of transactions monitored. Therefore there is a need for decreasing the amount of transactions to be analysed by a performance engineer. Angerstein et al. identified that oftentimes there are similar problems with similar or even the same root cause. For that matter, they try to group the performance incidents by aggregating them, for example with the k-means algorithm, making the analysis more efficient as the number of problems decrease.

An approach in the automatic analysis of APM data also deliver Heger et al. [35] with their collaborative project diagnoseIT. They try to improve the usage of APM solutions by automating the performance problem detection as the manual process is error-prone and very time-consuming. The focus is also laid on the identification of the root cause of performance problems.

3.2 GUI Test Case Implementation Strategies

This section covers different test implementation strategies than the one used in this thesis. Therefore they deliver either an addition to this approach or an extension for deeper, more specialized root cause analysis. Pradel et al. [52] present an interesting approach of automatic generation of GUI-Level tests. Based on JavaScript events an inferred finite state model of the user interface is used. The states correspond to a specific view which contains a finite number of JavaScript events that will switch the view and put it in another state. The algorithm then tries to identify tuples of events that slow each other down with each successive execution by randomly triggering events.

A problem still lies in the definition of the states as they are only identified on the document title and URL. As there are a lot of applications in the firm where the triggered events don't change the document title or the URL the algorithm couldn't identify a different state after triggering an event. Another possible problem is the complexity of the first algorithm which increases quadratically with events. Although the number of events is finite, there can be applications with such a large number of events that the algorithm's execution time significantly increases.

Ermuth et al. [23] present an interesting approach by combining micro-events such as clicks or mouse overs and combining them to macro-events. Therefore they deliver an extension of the above mentioned approach and optimize it by enabling the algorithm to cover more events.

The previously mentioned Automated GUI test case implementations have the goal to evaluate responsiveness problems and can be executed in addition to the approach taken in this thesis to further increase the software's quality and performance.

Another approach is the identification of workload dependent performance bottlenecks (WDPB) presented by Xiao et al. [76]. With the approach in this thesis, the problem of WDPBs is only covered for the number of users through the peak workload definition. The approach in this thesis doesn't consider single user workload-dependent issues during the execution of the use cases. Therefore the approach from Xiao et al. can be used as an addition.

Implementing realistic User Behaviour in the Load Test Script. Research is also performed on the thinktimes of performance test scripts. Ramakrishnan et al. [53] state that the need to implement the test cases with realistic thinktimes as the user needs a different amount of time for different actions during his visit on a webpage. Sometimes he needs to check his input in

a text file or he needs to search different entries in a list. To reproduce that performance test scripts use thinktimes between different user actions. Ramikrishnan et al. offer an algorithm to evaluate real user thinktimes based on webserver logs. If there is an APM tool available it would be possible to also extract these thinktimes automatically from it through the REST API. This is definitely nice to have for the performance test to generate more realistic load and possibly identify different bottlenecks than by normally or randomly distributed thinktimes. In this case study normally distributed thinktimes were used because of the lack of a productive environment.

3.3 Performance Evaluation

This section covers different aspects of performance evaluation excluding load testing which will be covered in the section 3.4. However it includes two techniques which help in optimizing software quality. Benchmarking can be used for subsequent load test executions and testing different global locations like in subsection 5.2.5 or testing different application configurations. The profiling technique used in this load testing approach is byte-code instrumentation from the APM therefore making it related to this thesis.

3.3.1 Benchmarking

A possible solution for performance regression testing would be the one provided by Heger, Happe and Farahbod [33]. The ability to identify possible regressions according to builds and methods would be really nice cost effective feedback. Modern APM solutions already provide an evaluation of the method, which is consuming most of the time and a possibility to compare different executions of the same method. Heger et al.'s method is closer to the developer which is also important as it follows the DevOps paradigm and delivers operations data to the developer. It is an addition to the approach in this thesis where we measure synthetic end-user response times and identify issues in the APM.

In the case study a geographic location benchmark is executed and analysed in subsection 5.3.3. Automated benchmarks are also executed between the subsequent deployments of the application through daily test executions to assess the relative performance between the possibly shippable product increments. This is described in subsection 5.2.4 of the case study.

3.3.2 Profiling

Profiling in this approach is done by the APM through a byte-code instrumentation at the start of the application and therefore the data will be used to identify the root cause of issues. It is configurable, as it is possible to instrument particular methods or components of an application deeper if needed by adding respective sensors.

Svogor [68] used execution time profiling and power consumption to determine, whether CPU, GPU or FPGA is more efficient in the software components namely image filtering and object detection.

Callan et al. [13] developed a profiling with zero overhead based on electromagnetic emanations. No instrumentation is needed for their profiling computations but a lack of accuracy is observed, in their example they profiled with 94% accuracy. The approach needs a time-intensive training phase with instrumented and uninstrumented code to map the executions

through the electromagnetic emanations. The APM used in this approach has monitoring overhead but the CPU consumed by the instrumentation is self-monitored to control the monitoring activities.

3.4 Load Testing

Although load testing is a part of performance testing in section 3.3 it will be covered in a separate section because it is an essential part of this thesis. In chapter 4 an entire load testing process is presented. Load testing is used to verify a system's capability of dealing with a certain load pattern making it a useful instrument to evaluate software quality.

3.4.1 Automatic Detection of Performance Incidents

The automated detection of performance problems based on load test results delivers Jiang [37]. He uses the repository of load test results in order to verify functional correctness and evaluate performance criteria automatically by generating respective models.

Grechanik et al. [32] created a dynamic load test approach, where they automatically modify the input data based on the previous results and are then able to detect other performance problems with the subsequent test executions. This is performed on GUI test cases created by performance engineers. Their approach can automatically detect and, to a certain degree, isolate performance problems. It is an extension to conventional load testing and focuses on different sets of input data.

Walter et al. [70] give an approach to evaluate SLAs during different stages of the software life cycle by validating them through model-based and measurement-based analysis.

3.4.2 Detecting Software Performance Anti-Patterns

There is research effort spent in the automatic detection of performance anti-patterns which is related to the topic of this thesis as it delivers another approach in detecting performance incidents in an early stage of the development. Peiris and Hill [50] present an approach of classifying and identifying such performance anti-patterns by only using system performance metrics, ignoring source code or application metrics. They achieve that with a non-intrusive machine learning approach but they don't mention the usage of an APM tool. If they would include such an APM tool in their analysis, not only could they use system performance metrics but also application metrics to a certain degree to further improve their approach.

Another approach in automatically detecting typial software performance anti-patterns deliver Wert, Happe and Happe [73]. They also require a test system and a usage profile and use them to uncover performance problems including their root causes in Java-based three-tier enterprise applications. They also create a "Performance Problem Hierarchy" to further classify the performance problems, which seem to be a common practice to minimize the effort of identifying performance problems in large-scale-systems.

There are a few special techniques that focus on specific systems or applications to detect performance antipatterns [65] [16] [17]. All these approaches deliver significant results for their posed problems but don't deliver a generic approach or only focus on identifying performance antipatterns on an architectural level. However it is related to the topic of this thesis and delivers another solution in detecting performance incidents during the development phase.

3.4.3 Predicting Performance with respective models

Woodsite et al. [75] define two major performance engineering approaches:

1. Measurement-based performance engineering
2. Model-based performance engineering

This section shows related work about the model-based approach.

There are performance engineering research studies about predicting the software's performance on different workloads based on collected APM data. Instead of manually analysing the system, these other approaches try to automatically generate a performance model or different performance models and predict the applications capabilities and response times based load test results. As Brebner [9] suggests, it is often useful to build multiple competing models to validate their correctness. He also states that if there is not enough information to buid one single "über" model it is possible to build multiple specialised models for different purposes. In comparison to manually analysing, the automatic generation of performance models is more efficient, both time and effort wise. It is also possible to calculate different kind of loads with the same datasets, if this is necessary. If the load can reliably be estimated, this advantage doesn't carry weight. With load testing you can measure the end-users true response times for single actions instead of only calculating the response time distribution. To summarize, load testing delivers deeper insights but is less scalable in terms of time and effort. Load Testing can also be used to verify the correctness of the predictions of the automatically generated models.

Brebner [10] presented 3 Projects where he tried to predict the response time distribution and the maximum capacity of the applications. He uses the APM's REST interface to extract the collected data which are necessary to build the model. Currently there is no standard representation of APM data and different vendors propose different representations, so this is a hard task to generalize and automate. One downside is the smallest error rate, which can be pretty high. Also the model's complexity rapidly increases with a larger amount of transactions monitored. Other problems also are the error rate of the load tests due to erroneous functionality. That is a disadvantage compared to the manual analysis, where these errors can be ignored and the analysis isn't stopped.

A similar example deliver Willnecker et al. [74] with a concrete example. They integrated Dynatrace in a Java EE application and generate a standard model (Palladio Component Models, PCM), which then can be analysed by existing simulation engines. In contrast to the previous examples, Willnecker et al. only create one model, which is error-prone in the case that it has specific requirements to be met for creating the model. PCM [49] is an open source software project, which provides the analysis of four quality dimensions (Performance, Reliability, Maintainability and Costs) based on input from the different developers. Research on this approach is supported by the German Research Foundation (DFG), for example the short overview of the PCM provided by Becker et al. [8]. In this paper the structure and the usage of the PCM approach is further discussed and evaluated based on a case study.

3.5 Different challenges in Web Performance

This section shows current research in web performance testing and engineering. It gives an overview on the current challenges and possible solutions to problems identified in web applications. The subsections are selected based on the challenges mainly faced in the financial institution and the case study. For example bandwidth is excluded because for internal usage

there exist ensured bandwidth. Latency is further discussed in subsection 5.2.5. Throughput is covered in section 4.4 and subsections 4.2.4 and 5.2.3. Scalability is not discussed deeply in the approach but can be covered through the infrastructure monitoring discussed in subsection 4.5. Since the case study implements JavaScript in the front-end which causes performance problems it is also included here and further discussed in subsection 5.3.4.

3.5.1 Latency

Since the company has offices spread around the world latency naturally becomes a problem.

In large-scale web applications, which should be accessible all around the world, there are different challenges in performances and specifically in Latency. According to Sundaresan [62], Latency is becoming a major performance bottleneck. The requirements on the user interfaces are rising as those are becoming more and more important for the applications. There need to be high quality pictures and nice page design which can lead to very high page sizes. If the page size becomes bigger the impact of the latency is rising as these large files need to go through the network. There are different ways to deal with large files e.g. enabling compression.

Another engineering aspect where the latency can become a bottleneck are the number of messages sent through the network. The number of messages sent through the network also have a correlation with the sizes of the files to be sent. Concerning performance engineering, the number of messages sent has to be minimized as for every message sent, the doubled latency (round-trip-time) has to be added to the end user's response time.

3.5.2 Throughput

The problem of throughput optimization for cloud-native relational database conquered Verbitski et al. [66]. They stated that after the optimization of their database structure to further improve their databases throughput they identified the network as the next bottleneck. This example shows, that in order to optimize throughput in an application, the performance bottleneck needs to be identified to maximize throughput. The problem of throughput optimization includes all of the response time contributors in an application. The methodology is heavily related to this thesis as the identification of the performance bottleneck can be achieved through load testing.

Setting realistic thinktimes, as Ramakrishnan et al. [53] propose, is also an instrument in end-user use cases to reach realistic throughput in a load test. However it doesn't deal with optimizing throughput in the analysis process, it only helps at the implementation and replication of realistic user behaviour. Throughput optimization for HTTP is described by Rodge et al. [55] in their work. They present tweaks in system settings to optimize throughput. These can be used additionally to this approach when identifying a throughput problem on communication basis.

In regard to the approach taken in this thesis throughput is an important metric. It shows the systems capability of dealing with a certain load which is a key metric describing the quality of software. Internal applications define the throughput

3.5.3 Scalability

Scalability can be partitioned in two major dimensions, as per Sedaghat et al. [57]:

1. Horizontal Elasticity: Varying number of VMs

2. Vertical Elasticity: Varying the capacity of VMs

Sedaghat et al. evaluated the cost efficiency of varying number of VM's and varying capacity of VM's by using price/performance ratio. They optimized the number and capacity of VM's based on the utilization cost. The scalability based on cost optimization will not be further discussed in this approach. The decision of how to optimally scale in case of a sizing issue detected during a load test is an addition to this approach. An example evaluation of vertical scalability of a database deliver Cheng et al. [14]. They increased the number of cores for two heave queries and showed, that at a certain number of cores a plateau is reached where the performance won't get better with the addition of more cores. They assumed that the reason for this was the message overhead.

As discussed in the previous subsection 3.5.2 it is possible to identify the performance bottleneck through load testing. The server infrastructure is one factor that can limit the throughput. If an increase of infrastructure is needed the decision in which dimension to scale up is needed. This makes the problem of scalability relevant in solving infrastructure related incidents.

3.5.4 JavaScript

The current state of practice in web applications was researched by Nederlof et al. [45]. They evaluated 4'211 randomly selected sites and concluded that:

1. 90% of the sites performed DOM manipulations after they are loaded, making the sites highly dynamic.
2. W3C standard violations and structural errors are widespread and have the potential to break applications.
3. Performance Related Guidelines are underused, 40% of the sites have blocking JavaScript's.

These findings show, that the client-side development potentially has room for performance improvements because of the dynamics, the error-prone development and ignored guidelines during the development process.

Ahn et al. [2] state that improvements in browser interface test suites can be achieved by focusing on the optimization of JavaScript. Although JavaScript has no concept of types, the Chrome V8 compiler defines types. In order to achieve performance improvements, they modified the compiler such that the definition of types by the compiler is performed 36% faster with 20% decreased allocated heap memory.

Mehrara et al. [41] state that the performance bottleneck in web applications. Client-side computation is preferred to avoid high network traffic and increase the responsiveness of applications. With this evolution, there is a need to optimize client-side performance as it is consuming more resources and therefore consumes more execution time.

Na et al. [44] confirm that JavaScript is getting more computation heavy. The reason for this can be identified with the introduction of the HTML5 standard. They also try to optimize the JavaScript performance by optimizing the compiler and extending it with parallelization.

In this approach it will be possible to identify the client as bottleneck, finding the root cause won't be possible. The research is also focusing on optimizing the compiler [2] [44] to improve performance.

This thesis executes the test cases on the browser API. Therefore the involvement of JavaScript related incidents can be a key factor during the identification process of client-side related incidents. This thesis doesn't dive deep in the incident identification process of JavaScript related incidents. However it will be possible to categorize an occurred incident as a JavaScript incident fast during the analysis.

Chapter 4

Approach

This chapter describes the general approach taken in this thesis. It covers the prerequisites for this approach, the test design, the test execution and the analysis of the executed tests. Hence it is a step-by-step description of the load test process focusing on browser GUI implementation and browser API execution. Possible solutions for optimizing single steps are elicited because the main challenge faced is the closer time constraint compared to the previous, big bang approach. These optimizations include test implementation, execution and analysis. The whole process is then integrated in a project in chapter 5.

4.1 Prerequisites of this approach

This section defines this approaches' prerequisites and briefly discusses, if these prerequisites can be met with the more agile approach.

Load Generator. For this approach, measuring the true response times of the end user to capture the whole user experience and enable a complete analysis, the load injected needs to be on the GUI level. In contrast to protocol level testing, this approach also measures JavaScript executions on the DOM-tree, delivering true user experience response times. This adds an additional dimension to be considered in the analysis which will be discussed in the chapter client-side performance measurements 4.4

APM as a monitor. For the server-side analysis, an APM is needed to monitor the test execution.

Dedicated SuT. For the performance test to deliver useful and representative results it is important, that these performance tests are executed on a dedicated environment, to decrease the impact of noise which would falsify the results of the executed performance tests. If there is an APM integrated in the environment to be tested, occurring noise can be identified when observing the systems behaviour (e.g. web requests, cpu consumption, memory usage etc.) before and after the test run. If noise is identified on the dedicated SuT it has to be removed. Otherwise the results of the performance tests are falsified and can't be used to reliably determine the software's quality.

4.2 Design of the Load Test

The design of a load test contains the definition of the performance relevant use cases, the different types of load patterns to run for the SuT and the workload calculation. For subsequent test executions in the development phase the design of the load test is done at the beginning. Hence this section is showing the advantages in detail to answer a part of research question 2 in section 1.2.

4.2.1 Use Case Definition

The test case definition for applications in the financial sector are composed out of business workflows. They have a clear structure and test data requirements. The test data can contain user credentials and can additionally have other fields like portfolios and instruments. The structure includes the single steps during the workflow. Each click and input is predetermined by the business workflow.

Business Workflow. Information about the business workflow for an application are gained from business representatives which are the future users of the system. An excerpt from the case study with measurements included looks like the following:

1. Type the url in the browser.
2. Start measurement 'M1'.
   (a) Press Enter.
   (b) Wait for Login Screen.
3. End measurement 'M1'.
4. Type your credentials.
5. Start measurement 'M2'.
   (a) Press Login.
   (b) Wait for the portfolios to be loaded.
6. End measurement 'M2'.
7. Type '123456' in the portfolio search box.
8. Start measurement 'M3'.
   (a) Click on search.
   (b) Wait for portfolio '123456' to be displayed.
9. End measurement 'M3'.

The data to be typed in the performance test is denoted as test data.

Test Data. The test data needs to be created from the application developers. Optimally, the test data consists of real production data. For the financial sector however, data security is a key factor and the usage of unencrypted real user or consumer data is not permitted. As the encryption and decryption negatively influences the performance, the usage of anonymized real user data is needed. Test data is also an important source of performance problems. An example occurred in the case study in subsection 5.2.5. An additional test run needed to be created to test the system based on the test data input scalability. The impact can occur on both client-side and server-side. In the client-side it can cause a memory overflow or long interface loading times. On the server-side it can also cause large database loading times or slow executions of certain methods.

4.2.2 Types of Load Patterns

There exist different types of load patterns in web performance end-to-end load testing to determine an applications quality in regard to performance. In the following section the different types of load patterns will be described which will be used for the case study. There are other types like a stress test that won't be covered in this thesis. First the description of the two general types increasing load and steady-state are described. The steady-state load then can further be distinguished between regular load, peak load and long running load. For illustration purposes the load graphs from the case study are used. The case study is deeper discussed in chapter 5. The detailed explanation of the abbreviations can be found in subsection 4.2.4.

Increasing Load. In an increasing load test, the load generated will be built up slowly, starting with a small amount of transactions. This load is used to determine the point of failure for the application. If this point is reached, we can identify the bottleneck of the application through the APM and therefore gain more knowledge about the limits of the application. An illustration can be found in figure 4.1. The duration of an increasing load is not determined by time. The end of an increasing load test is reached when a predefined amount of transactions fail due to unresponsiveness of the SuT. Unresponsiveness can either be defined through reaching a response time cap or by the number of http errors per second received from the SuT.

The numbers used in the following explanation are further described in subsection 4.2.4. The speed on which the transactions per minute are increased should be chosen conservatively because the point of failure can be evaluated more precisely the slower the increase happens. The point of failure is expressed through the amount of transactions per hour executed. The increase of transactions per hour is controlled by the addition of virtual users executing use cases with the same goal session time. To evaluate the exact point of failure the virtual user addition speed needs to be adjusted based on the transaction execution time. Let the first virtual user finish one transaction then add another virtual user to the test.

Theoretically it is possible to decrease the GST instead of adding virtual users. It would enable a more precise evaluation of the point of failure as the GST is a floating number whereas the number of virtual users is an integer. However it falsifies the real usage of the application as the goal session time is an evaluated value which inherently includes realistic thinktimes. Also by increasing virtual users instead of GST it is possible to determine the point of failure not only through the maximum amount of transactions but also through the maximum amount of users.

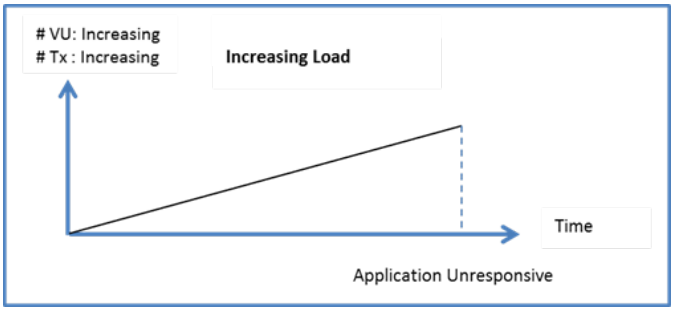

Figure 4.1: Increasing Load

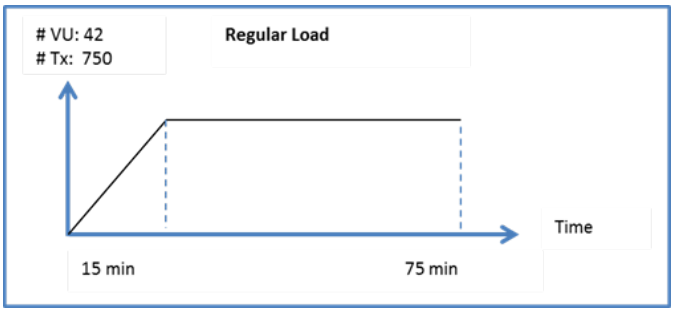

Figure 4.2: Regular Load

Steady-State. In a steady state performance test, the load generated stays the same throughout the whole execution. It can contain ramp-up and ramp-down times. A ramp-up is done at the start of the test, where the number of transactions per hour are slowly increased until the defined steady-state load is reached. The ramp-down is the respective opposite, where at the end of the performance test, the number of transactions per hour is reduced until it reaches 0. It is also possible to define a warm-up time where no measurements are taken. The steady-state load can further be refined in the next three workloads. In this approach we don't have a warm-up time. However the ramp-up time is statically set to 15 minutes.

Regular Load. The regular load test is executed to validate the SuTs capability to deal with regular day in usage. An illustration can be found in figure 4.2.

Peak Load. The peak load test is executed to assess the SuTs ability to deal with a higher amount of transactions than the regular load. These higher loads occur on special occasions like a presidential election where the trading of financial products heavily increases. Schematically it looks the same as the regular load depicted in figure 4.2 only the amount of transactions per hour and virtual users is increased.

Long Running Regular Load. A long running regular load test is used to determine the applications behaviour over a regular business day (usually 8 hours). The focus for this test doesn't lie in the end-user response times but rather in the behaviour of the infrastructure. If there is a possible memory leak it is possible to identify it with a long running regular load test. It is also able to identify client-side memory issues when multiple sessions are executed subsequently. Graphically it is the same as the regular load depicted in figure 4.2 only with a longer maximum running time.

4.2.3 Selection of Load Patterns

The selection of a load pattern can differ for each application. It is dependent on the critical performance factors of each application and what the goal of the performance testing activities are. For example in our sample case study of the portfolio risk calculator, the testing of the standalone software from the external provider was focused on meeting the internal requirements on this part of the software. It was important to gain knowledge about the capabilities and boundaries of this software and if the portfolio risk can be calculated in a certain amount of time. So we decided to implement different types of increasing load as input (increasing amount of portfolios, increasing amount of positions, increasing amount of parallel requests).

In other cases, it is more suitable to verify the capability of the software to deal with an estimation of a regular daily load. This is especially the case, if the application is already deployed and in productive use as we then already have the number of users and transactions which the application needs to deal with. Steady-State selection in the workload definition would be the right choice in that case.

If there is a suspicious growth of memory consumption observed in the regular load test, indicated by a constant growth of old generation memory increase with or without any garbage collection activities, a long running regular load test is an ideal way to verify a possible memory leak.

4.2.4 Workload Calculation

The definition of the workload for end-user use cases consists of three measures which correlate with each other. They need to be defined for each load type.

- Transactions per hour (Tx/h)
- Goal Session Time (GST) in Seconds and including the thinktimes
- Number of Virtual Users (VUser)

The goal session time determines the execution time of one use case to reach the wanted throughput. The correlation of the three measures is as follows:

$$Tx/h = \frac{3600}{GST} VUser$$

In order to achieve a wanted throughput in the form of the number of transactions, it is possible to either vary the GST or the number of virtual users. This can be useful, if you don't have enough testing infrastructure to simulate a large amount of virtual users.

As for internal applications with a finite number of users, the definition of a regular day workload can easily be done. When a system already is in productive usage, the regular day workload can be observed. If it is a new system, the workload has to be evaluated. High peak usage of an application, for example on special occasions like a presidential election, is tested with the doubled amount of transactions than the regular workload if no production data is available. Otherwise an analysis of the usage over a year is needed to determine the load generated on peak usage days.

Calculating the Workload. The calculation of the workload starts with the evaluation of the user group identifying the number of users. To get the number of transactions on a regular day, a business representative needs to be contacted. The number of transactions naturally comes by the business requirements regarding the number of business transactions executed. By multiplying the number of users with the number of transactions per user the daily number of transactions can be determined. This can be mapped with the desired execution time of the performance test. In comparison to the previous approach (non-agile) there is no difference in calculating the workload. When transferring from development to production the workload can be redefined based on the productive monitoring data. The need for continuous verification of the defined workload arises in the agile approach. Therefore regularly validating with current productive data is necessary.

4.3 Test Implementation

In agile development the length of the cycles where a shippable product increment is created is not fixed. It can be reduced to a minimal amount of time as elicited in section 1.3. This poses a challenge in the test implementation. The discussion in reducing time for the test implementation includes a brief discussion of using functional test cases and reducing the time spent for maintaining the test implementation in subsequent deployments.

4.3.1 Using Functional Test Implementations as Performance Test Cases

A resource optimizing choice would be the usage of test cases, which are already maintained by the project team as Ferme et al. [24] already suggested. This section covers a brief discussion about using functional test cases for the performance test. Note that in this case, unit tests are not included in this evaluation as they only execute fractions of code. Additionally they do not follow rigorous performance evaluation strategies which is depicted by Georges et al. [29].

Advantages. The usage of common test cases for functional and performance tests saves implementation time of the test cases and ensure the maintenance of them. Especially considering the fact, that the application is changing in every deployment and therefore increasing the probability of a change in the test case requiring the tester to update his implementation of the test. If this has to be done for functional test cases and performance test cases separately, the overhead of effort increases.

Disadvantages. In practice the common usage of test cases can be difficult, as there are different tools in use for functional and performance tests. If there are already two different, independent teams and tools for performance and functional testing, it will be hard to combine

the test cases. Another disadvantage depict the differing quantity of test cases for functional and performance tests. Functional testing needs to cover a large number of test cases whereas performance testing focuses on fewer cases with an actual performance impact. Therefore a selection of test cases with performance impact would be needed.

Summary. In financial applications with a predefined workflow and known use cases, the usage of functional test cases is currently not needed. However if in the near future the deployment cycles further decrease in time, the combination of the test cases could save time for the maintenance, when the cases differ between deployments. To achieve such a state, the different testing teams need to have compatible tools which will be an obstacle to be surpassed.

4.3.2 Efficiency and Maintainability of the Test Implementation

Motivating example. Imagine John Doe implements a use case for the first time. The effort spent for the initial implementation is high compared to subsequent release executions. This is because the navigation, the element to be validated and the measure starts and stops need to be scripted (depicted in section 4.4 and subsection 4.2.1). In the next release an interface change is implemented by the developer. This needs an adaption in the test script because the DOM properties or even the DOM element type could have been changed. Therefore the tester needs to update the navigations and the elements to be validated. This is additional implementation effort that needs to be spent in order to successfully execute the performance test. This can take a large amount of time to achieve as John always needs to revisit the respecting screens in the GUI to update the test script. In order to decrease this implementation effort some efficient methods are presented in this subsection. As this also is a recurring task every two weeks it is worth investing time to optimize the effort spent.

Choosing an efficient XPath locator. As we navigate the use case through the DOM-tree in a browser, changes in the DOM-tree require changes in the script. Therefore, the selection of the identificators for DOM elements need to be chosen as generic as possible. An example for this fact is the following wait for function, where the identifier of the DOM element only contains the element node H1:

```
BrowserWaitForProperty("//H1", "textContents", "Add Transaction")
```

In contrast to the identifier with a property added to the node, this identifier is more robust to changes in the GUI. A change of node properties will not affect this wait for function making it less vulnerable to interface changes and if there is a change of this type of element, the update only requires the information of the new element node. However a lot of DOM nodes can have the same type and for the identification, a unique property is needed. It is important that the developer implements unique properties for important nodes. In the following examples and possible solutions it is assumed, that the developer implemented the unique property 'id' somewhere in the DOM-tree, not necessarily on the target node.

DOM elements can contain a lot of different properties which aren't good choices for the identification. For illustration we can take a look on table 4.1.

In the selection of the best property we have to consider it to be as unique as possible. We can generally exclude boolean values and integers as they won't be unique. If we also consider the

Property	Value
aria-level	0
aria-posinsert	0
aria-setsize	0
autofocus	false
class	button
contenteditable	inherit
disabled	false
draggable	false
hidefocus	false
id	splashCloseButton
spellcheck	false
tabindex	0
textContents	OK
type	submit

Table 4.1: DOM example properties

maintainability aspect, we can also exclude 'textContents', as button names or text in general is likely to change between releases for example due to cosmetic reasons. This leaves us with the properties class, id and type. In this example, choosing the 'id' as DOM identifier is the best choice in consideration of uniqueness and maintainability. However there can be cases where the id isn't the ideal choice. Take a look at the example in table 4.2 of DOM properties and values.

If the 'id' itself contains numbers that look like an enumeration as in this example, the' i'd is very likely to increase by 1, or modified in another way, for the next release. In this case the better choice would be to take the class as DOM identifier for the higher grade of maintainability.

As the DOM elements are ordered in a tree structure, it is also possible to choose the parent or child nodes as a locator point. An example locator where the parent node is used with an identifier to reach the desired subtree is the following:

```
//DIV[@id='portfolioListZone']//H3
```

With the same example we are able to show the possibility of using a sibling as the first locator if there isn't another node with a maintainable and unique identifier:

```
//DIV[@id='portfolioListZone']/..//H3
```

This can be useful if the desired object and the parent node don't have unique and maintainable properties for the identification. Another advantage in regard to performance is the usage of absolute paths (denoted with '/') in contrast to the relative path (denoted with '//'). Using the relative XPath has a negative effect on the response time. It can cause a delay in the range

Property	Value
aria-level	0
aria-posinset	0
aria-setsize	0
class	portfolio-list-area home
contenteditable	inherit
disabled	false
draggable	false
hidefocus	false
id	p123456
spellcheck	false
tabindex	0

Table 4.2: DOM example class vs. id

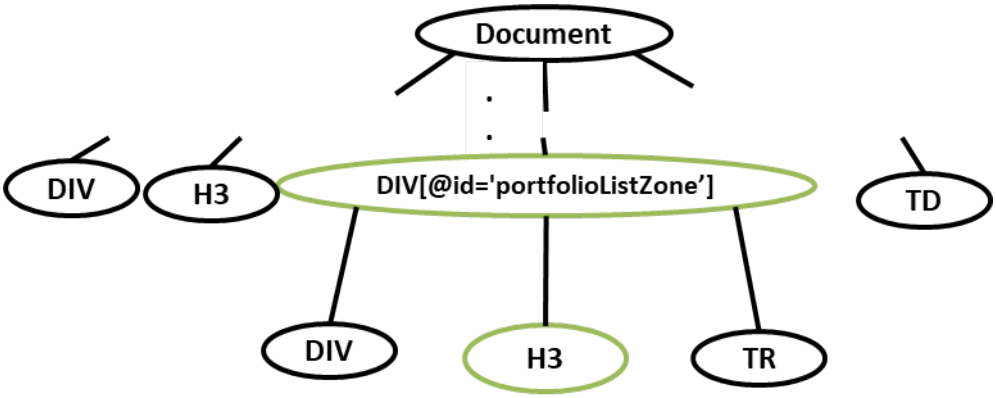

Figure 4.3: DOM tree example parent-child

of milliseconds in finding the desired node.

All of the above mentioned decisions about choosing the right property for identification can be applied to the property to be validated with the exclusion of using other nodes than the target node.

The idea for the following algorithm is from me, the author of this thesis. To the best of my knowledge this algorithm was not published in this context by anyone else.

Proposal for Automating the Selection of the DOM Identifier. With these operations you are able to navigate the whole DOM-tree from top to bottom and from bottom to top. Therefore you are able to find ideal properties for the locator. It is even possible to automate this process with basic tree operations. In order to achieve the shortest path from the first locating node, you could use this node as a starting node for the breadth-first search algorithm. The usage of breadth-first search is needed in this approach as this search first looks for the immediate child nodes. After the first child level search of this node, check the parent node as it has the same distance as the target nodes children. With these two steps, we covered all the nodes with

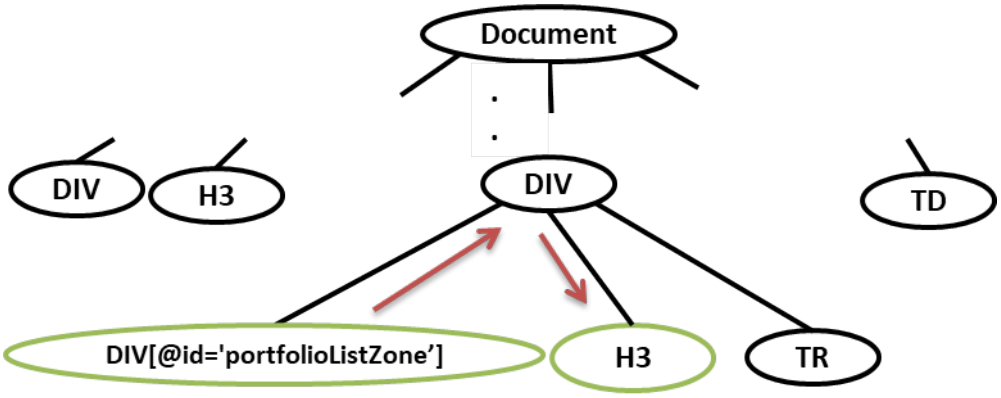

Figure 4.4: DOM tree sibling example

distance 1 to the target. The next step is the merge of the target node and all the nodes within distance one to the a new node restarting the algorithm at the beginning. Note that the steps 1 and 2 can be switched. The graphical illustration can be found in the figures 4.5 and 4.6. To summarize the algorithm in four steps:

1. Breadth-first search on the targets immediate child nodes
2. Check the target's parent
3. Merge the target node, its immediate children and its parent to a new node
4. Treat the new node as the target node and start from step 1

With this algorithm you are able to find the closest node (in regard of number of nodes passed through) that has a unique and maintainable locator. As elicited before, the search criteria for the node properties should be chosen as the id of the node with no numbers in it.

Creating Functions for Recurring Test Steps. In order to keep the test implementation maintainable it is necessary to generalize recurring test steps. Especially in test suites with multiple use cases in the same screen it is desired to not have duplicated XPath locators. This is possible in a utilities file of the test suite where the recurring test steps need to be located. For example in our case study the addition of multiple instruments was needed. In a regular test script that would be achieved by copy pasting the test steps and therefore multiple times the same locators. If they then change in a subsequent release you have to change this locator on three different lines of code. The creation of a method called 'addInstrument("Instrument Identifier")' is desired because we don't create XPath locator duplicates making the test script more maintainable.

However if you outsource test steps in this way you have to keep in mind that the add instrument operation can have different execution times for different instruments added. If this is the case the timer name also needs to be modified with some kind of identifier to keep them separated. In our case the timer name would be added with the instrument identifier.

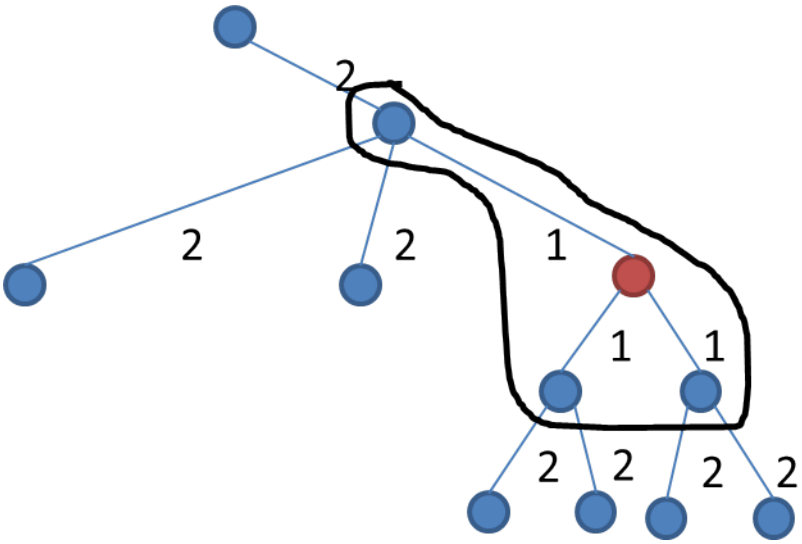

Figure 4.5: DOM tree search step 1 and 2

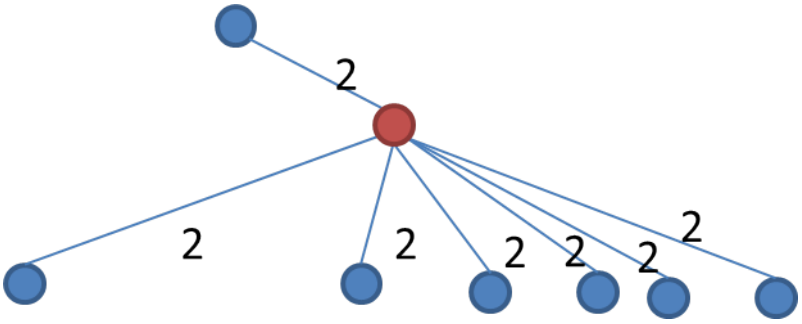

Figure 4.6: DOM tree after the merge of the nodes with distance 1 to the target node

4.4 Client-Side Performance Metrics

The client-side performance measurements include measuring end-user response times. These are then further aggregated to performance metrics further discussed later in this section. The end-user measurements are taken by the following scheme:

```
MeasureStart("UC1_TimerName");
   BrowserClick("XPath-locator-to-a-button", BUTTON_Left, "TimerName");
   BrowserWaitForProperty("XPath-locator-to-a-DOM-element", "Property-to-be-validated",
      "Expected Value of the Property");
MeasureStop("UC1_TimerName");
```

Listing 4.1: Example Time Measurement in the Test Script

The timer is identified with a string, the prefix stands for the use case to have a better overview in test suites with a large number of use cases. As the navigation happens on the browsers DOM-tree, XPath expressions will be used to locate the desired nodes. These are the first parameters in the click and the wait for function. You are also able to right or left click with the second parameter in the click function. The third parameter is used to associate the click with the corresponding timer name. This parameter is also used to later identify the web requests in the APM that were executed after the click (described in subsection 4.6). Amongst the XPath expression, the wait for property function also contains a validation property name and the expected value. After the specified DOM node in the wait for function is displayed in the browser, the measurement is stopped.

The 'BrowserWaitForProperty' function is necessary as there can be asynchronous requests that aren't triggered by the click. For example subsequent JavaScript executions after the click can cause subsequent requests to the server. These requests then aren't measured with the 'BrowserClick' function. With the wait for function we are able to measure the whole response time.

In the following paragraphs, the client-side metrics that are captured during a load test run are presented. Of course there are more metrics like the number of http requests sent or the number of bytes sent. These are used for the in-depth incident analysis and are also manifested in the metrics captured. The thresholds for the metrics need to be defined according to the size of the generated load. Increasing load tests for example naturally have a high error rate, as this is supposed to happen and a regular load test has lower response times than a peak load test.

Response Times. The previously introduced time measurements are implemented in the test script following the performance relevant end-to-end use cases. Each response time measurement is saved and the aggregated metrics are then further used for the analysis. The aggregated metrics used for the initial analysis include:

- Average Response Time
- Standard Deviation
- 90^{th} Percentile
- Maximum Response Time

The thresholds usually are defined on the average response time and the throughput. The standard deviation is used to check the deviations of the measurements for a test run. The 90^{th} percentile can additionally be used to analyse deviations between test runs when benchmarking

an application. The maximum response time can also be an indicator that there are very few outliers in the collected measurements which aren't represented in the standard deviation or the 90^{th} percentile. If the maximum response time highly deviates from the 90^{th} percentile, higher percentiles need to be analyzed in order to check on the number of outliers. It is hard to evaluate which metrics are important for the analysis of a load test. For example it is also possible to include other metrics such as different percentiles. In order to not have too much numbers to analyse this approach only includes the listed metrics above for the initial analysis. The choice of which percentile to include lies in the definition given by the applications requirements. They decide on which percentile they want to have a performance assurance.

Errors. The error rate of a load test is observed by the number of transactions failed and the number of transactions passed with success. In this case a transaction is equal to one use case. Errors can have different root causes. Through the virtual users perspective there are four types of errors that can be observed:

1. HTTP Errors: Errors caused by the back-end.
   (a) 401: Can occur if there is a change in the user setup and the test user doesn't have access to the application anymore. Especially during the development process, this can occur when executing automatically triggered load tests.
   (b) 500: Oftentimes occur during high load tests where the back-end can't deal with the large amount of transactions generated through the load test.
2. JavaScript Errors: Errors on the GUI
3. Validation Errors: Occur, if there are test script issues. These can be fixed by updating the test script or making it more stable regarding XPath locators.
4. Test data Errors: Occur when using wrong test data. They usually can be observed in an error message generated by the application itself.

Throughput. The throughput is defined by the number of transactions successfully completed. This metric has strong correlation with the response times and the error rate, as both of them influence the throughput. If the response times are slow, the goal session time can't be held and the throughput isn't reached. If the error rate is too high the throughput can't be reached because not enough transactions successfully complete.

4.5 Server-Side Performance Metrics

In this approach we are using Dynatrace [21] to gain server-side performance metrics. It is able to monitor applications during runtime through byte-code injection at the start up [22]. The agent, which is installed on the SuTs webserver, sends the bytecode to the collector where it is modified according to the defined instrumentation. There is also an option to monitor client-side metrics in the APM. Since we are measuring the client-side metrics through the load generator this feature is not used in this thesis.

Infrastructure. The infrastructure monitoring covers the CPU consumption, the memory usage, the network utilization and the disk I/O on the web server. The APM automatically plots the infrastructure data over time in charts simplifying the observation of changes during runtime. You are also able to see the activities on the webserver before and after a load test, ensuring the idleness of a webserver and the behaviour during the load generation and afterwards. The different instrumented processes on the web server are also made visible and can be used for deeper analysis.

For deeper analysis on the CPU consumption in percent and memory usage, the APM offers the capturing of snapshots for the CPU, memory and threads. They can be taken at any time for deeper analysis of incidents. A full memory snapshot shows all the data from the heap showing the classes, objects and references enabling the identification of space consuming structures. The thread dump shows the currently running threads for an agent and their CPU time consumption. The CPU snapshot shows the idle time and busy time including the instrumentation overhead (in seconds) from the agent. However these snapshots need to be taken carefully as they increase the monitoring overhead.

Data storage. The data storage monitoring shows the time consumption for the instrumented database enabling the identification of time consuming queries throughout a load test. If the precondition of an idle environment is fulfilled, the identification of frequently executed queries is possible during the regular load test. These are opportunities to efficiently optimize the databases performance with high impact.

Execution Trace. The execution trace in Dynatrace is called PurePath. It shows the method calls and their time consumption including the database queries executed and the exceptions occurred, if any. For each executed method, the Dynatrace agent is collecting the following properties:

- Execution Time
- Breakdown: % of Execution time spent for CPU, I/O, sync, wait or suspension
- Class
- API
- Agent where the code is executed
- Thread Name

With these collected data you are able to identify high execution time contributors for a PurePath. Through sorting the PurePaths in descending order, you're able to identify the slowest requests sent to the webserver during a load test for further analysis.

Errors and Exceptions. The bytecode injection also traces uncaught exceptions, HTTP errors, log messages and browser errors. Again with the precondition of an idle SuT the count of exceptions and errors can be shown for the executed load test. This is useful to show the improvements or decrease between different benchmarking runs or test executions between different releases during a development process.

Transaction Flow. The transaction flow gives a summary of the horizontal transaction flow of the instrumented webservers and all the different entities . The response time contribution of each entity is summarized and shown as percentage response time contribution. Additionally, the number of calls per minute is shown between the different entities to give further indications of possible n+1 problems [51]. It is possible to view a single PurePath transaction flow or give a complete overview of a performance test run.

Fine-Tuning of the APM: Trade-Off between Overhead and Granularity. The monitoring overhead is dependent on the configuration of the APM on the granularity of your monitoring activities. An example for the configuration of the APM needed is the instrumentation of the memory sensor and creating automatic snapshots of the whole memory. This is a configuration needed if you have indications of a memory leak but it also increases the monitoring cost and can therefore have a negative impact on the applications response times. So there is a trade-off on how much monitoring is needed to identify the issues you want to exploit. This is especially important if you have the APM integrated in the production environment. In order of achieving the optimal configuration, load and performance testing is also an essential part.

The usual process in achieving a good enough monitoring granularity is by doing it top-down starting with a default configuration. This should be chosen to have as less impact on the performance as possible. If you find performance issues, where the root cause can't be identified with your current APM configuration, you can selectively modify it to finer granularity. Always keep in mind, that this modification in monitoring granularity can have a negative effect on the applications performance but in order to finding the root cause of an issue, this is sometimes needed. In the retest you can verify the performance issue and also observe the added monitoring overhead. If the impact of the monitoring overhead doesn't have an impact on the applications performance it is recommended to keep the new configuration if it helped identifying the issue. Therefore you are able to monitor this issue in the productive environment.

Over time there will be multiple APM configuration changes which raise the monitoring overhead. The monitoring overhead can reach critical amounts which significantly influence the performance. This critical amount is individual for each instrumented tier and has to be individually evaluated. If a critical amount is reached it is important to be able to see all the modifications you have done and undo the unnecessary and solved ones. Therefore it is important to keep track of the monitoring modifications and possibly prioritize them on the impact and their benefits on securing the software quality. The same problem face Heger et al. [35] in their diagnoseIT work. They stated that they want to focus on adaptive instrumentation and also efficient techniques to manage trade-offs between measuremet detail and system perturbation.

4.6 Observation and Analysis of Incidents

With the proposed approach we are able to correlate the user actions with back-end processes which is key in categorizing or even identifying the root causes. The incidents in this case are defined as observed slow response times or errors. These errors can also be caused by test implementation incidents. In our case these incidents are identified during the analysis process.

Correlating Client-Side Metrics with Server-Side Metrics. The correlation between the virtual user actions and the started back-end process on the webserver happens through a HTTP

header information. An additional field is added to the request sent from the browser to enable the identification in the APM. The field contains three string values:

- Scriptname: The name of the test script.
- Timername: The name of the started timer.
- Virtual Usergroup: The name of the virtual user group from which the request was sent.

This information is then added to the according PurePaths in the APM therefore enabling the correlation between user interaction and the triggered back-end processes. It enables the analysis of single user actions, separate use cases or the whole test run. The virtual user group is an additional implementation variant in the test script. It can be useful if the simulation of different types of users is necessary, for example if there are two different sets of test data you can implement two different user groups and analyse them separately in the APM. To visualize this correlation take a look at figure 5.1. The load from the client/load generator contains this additional Dynatrace header in each request sent from client to the SuT. This information is then passed through with the monitoring data to the Dynatrace collector.

Analysis of Single User Actions. The analysis of single user incidents during a load test can have two different observations. Either it is a response time which is out of the norm or an error is observed. As previously elicited, an observed error can be caused by a wrong test implementation or an SuT issue. The application issue is further distinguished between a client-side error or a server-side error. The client-side error needs to be further analysed in the browser. The server-side error and the observed extraordinary response time can be further identified and analysed in the APM with the previously introduced correlation including the timer name and the timestamp.

The analysis of an observed slow response time can be formalized to a certain extent in order to isolate the root cause. The formalization approach can be found in figure 4.7. It is a decision tree where the response time contributions are used as decision rules. At first the comparison between the measured end-user response time and the respective server contribution is done. This categorizes the incident into either a client-side incident or a server-side incident. This distinction is important as the further analysis is highly dependent on it. For a client-side incident the browsers' developer tools are used for a further categorization. The further client-side analysis is described in subsection 4.6.

The server side analysis with this approach is done with the APM. To further categorize the incident the most time consuming tier needs to be evaluated. If this tier is a web or application server the analysis of the respective PurePath is necessary. If the database tier is the most time-consuming the queries executed need to be further analysed based on their execution times. If another subsystem is identified as the most time consuming tier the analysis starts again with on the categorization for a server-side incident.

Analysis of a Test Run. Through the correlation between the client-side user interactions and the server-side processes it is possible to analyse the virtual users average response times and the server-side processing time and compare them. If there are significant differences between these two metrics, the root cause of the incident lies either in the communication or on the client. With this fairly simple analysis we can already narrow down the search for the root. The deeper analysis on the server-side is equivalent to the single user action analysis.

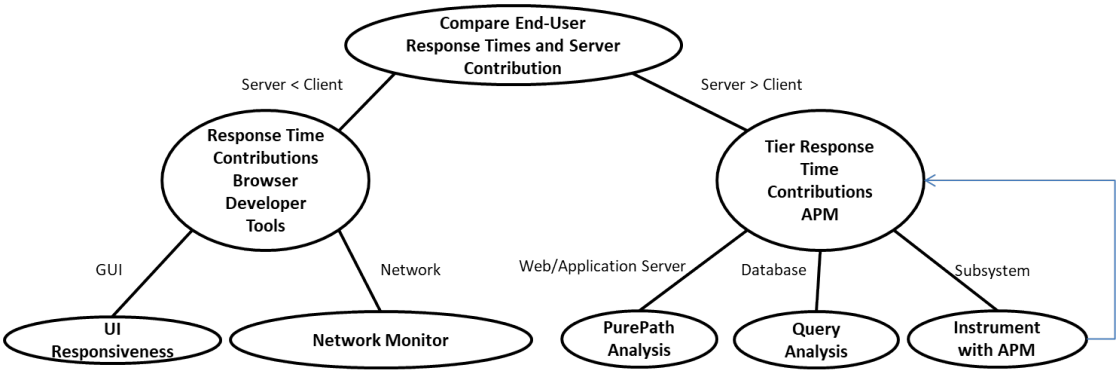

Figure 4.7: Formal Analysis of a Response Time observed by the End-User

Analysis of Client-Side Incidents. If a client-side incident occurred the analysis starts in the browser using the browsers developer tools. Turning on UI responsiveness monitoring further narrows down the identification of the root cause. It is possible to identify the most time consuming action in the browser. With the developer tools it is also possible to identify client-side memory issues. Through the creation of a browser memory dump the identification of the most resource intensive objects is enabled. Another useful function is the network analysis tool. You are able to identify the number of requests sent and the size of the requests which can be root causes for slow response times. This analysis happens manually through executing the user action that caused the incident.

Of course there are other tools that can be used to analyse client-side incidents.

User Experience Management (UEM). This is an addition to the APM solution. It can monitor end-user behaviour and also includes JavaScript injection. The nice thing of this solution is the real-time analysis during the load test and there is no manual execution of the test step needed. Although it delivers the same functionality as the browser developer tools, the analysis in this thesis is showed for the browser developer tools for illustration as the UEM option is not freely available.

Pagespeed/YSlow. Google with Pagespeed [31] and Yahoo with YSlow [77] deliver analysis tools for evaluating the performance of a web page. They include their rules of best practices and statically analyse the website. Amongst other things, the rules include server compression, caching options and compression of images.

Chapter 5

Case Study: Integrated Advisory Portal

This chapter describes the load testing approach for a project inside the company. The project is an merge of two different applications, which were already in use in the company. The client suitability tool delivered recommendations about the investment choices for a customer for certain portfolios and instruments according to his individual investment profile. The second application contains all the information about the portfolios and instruments therefore delivering the business data for the client suitability tool.

As the backend logic will stay the same in the new application, the previous two applications will communicate with each other through microservices. A completely new user interface is also part of the development.

The application team deploys a new release every two weeks on a dedicated test environment where the performance testing will happen. The revision of the GUI will be performed increasingly holding on to the business workflow, in other words starting from the first step and finishing at the last step. This enables the end-to-end performance test to evolve during the development process.

5.1 Overview

This section is used to give a brief overview over the SuT. It has various interfaces to other systems, which won't be discussed in detail in this section.

5.1.1 Architecture

The SuT is set up as a three tier .NET application. The tiers are as follows:

- Web Server with 2 Processes:
  - Case Manager
  - Investment Dashboard
- Application Server Case Manager
- Database Server

The connections to the other systems are built up during the development phase, starting with only mockups and slowly attaching the other systems. Through our use case all these tiers are set under load.

5.1.2 Performance Aspects

As all the tiers are set under load through our test, the response time contribution of each tier is an important metric in finding the bottleneck of the SuT.The response time contribution of each tier can also be used to identify the most performance critical tier where an optimization would have the biggest impact on the performance. It can also be a good starting point for the analysis of an incident as we efficiently narrow down the problem to the respective tier where the root cause most probably lies.

5.2 Test plan

The creation of the test plan contains the definition of the performance relevant use cases and the workload definition including the selection of the load patterns.

5.2.1 Use Case Definition and Implementation

The use case definition in the integrated client suitability tool follows a straightforward workflow based on the business' requirements, therefore we will only have one use case. However the challenge in the use case and implementation lies in the changing GUI through the development cycles. The use case needs to be redefined according to the changes and additions in the user interface. Every two weeks there is the need to revision the use case and adapt the test script to the GUI. In order to decrease the updating efforts, the test script needs to be implemented as maintainable as possible. Updating test scripts could negatively affect the response time. This can only be identified during the analysis.

Test Data. Input data for the use case consist of two different data sets, the test users and financial instruments to be added. This list and the access rights for our test users are provided from the development team. Additionally, the selection of the portfolio is done by iterating through the portfolio list in the interface.

5.2.2 Test Infrastructure

The testing infrastructure is as depicted in figure 5.1: The end user results are collected and accessed on the Silk Performer Master. The APM monitoring test results are collected on the Dynatrace collector and can be accessed through a fed client. For this test execution, 5 load agents are needed to simulate the 42 virtual users. The WAN-Bridge test can only be executed with one load agent because of infrastructure constraints.

5.2.3 Workload Calculation

Number of Concurrent Users. The calculation of the number of concurrent users is made data concerning the usage of the application

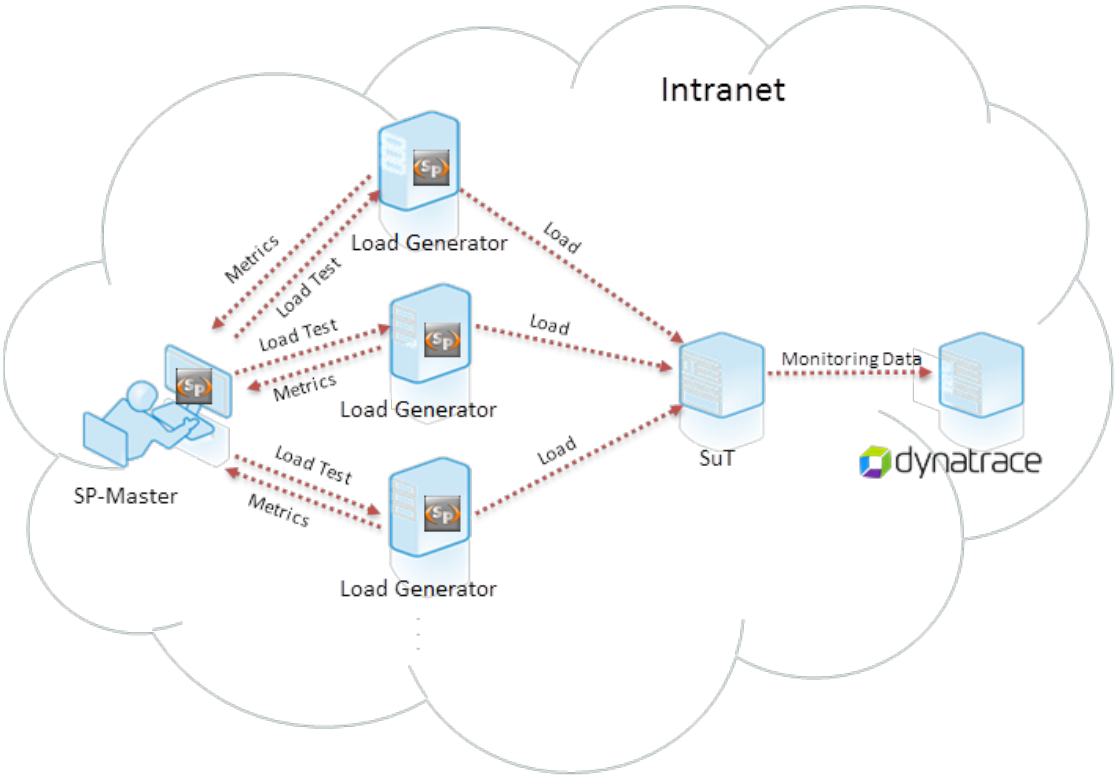

Figure 5.1: Testing Infrastructure Setup

- C = average number of concurrent users
- n = number of users accessing the system
- l = average length of usage
- t = time period of concern = working hours in minutes

The formula is a heuristic approach taken in the financial institution. It is as follows:

$$C = n*L/T$$

The employees which will access the application are predefined by the number of users using the previous applications today. In our case, we can start calculating with 1500 users per day. The average length of usage is also predefined to be 15 minutes. The time period of concern is a regular workday, 9 hours or 540 minutes. If we insert these numbers in the formula, we get the average number of concurrent users per day, rounded up to 42, which we will use for our regular workload.

Number of Transactions. The business usage evaluation shows that there is 1 use case per hour per user as the peak number of transactions. This number of transactions will be used for the peak load test. The regular load therefore is defined as half of the number of transactions as the peak load, 750 transactions per hour.

Selection of the Load Pattern. To validate the regular day usage of the system, the steady-state load is executed. The ramp-up time will be used to simulate realistic usage scenarios, as it is very unlikely that all the users will start using the application at the same time. The peak workload with the doubled amount of transactions will also be executed with a steady-state load with warm-up time. To verify the throughput of the system, the steady-state tests will run 1 hour with an additional 15 minutes ramp-up time.

5.2.4 Test Execution Schedule

The regular load test will be executed daily to get a performance baseline. The daily execution also helps in validating the test script after the deployment of a new release as there will be changes in the GUI. The changes can also affect test data problems. An example for this was a change in the instrument setup on a new release. Some of the instruments provided by the application team weren't compatible with certain portfolios anymore causing the failure of half of the executed use cases making the test invalid. In another release, the user setup was updated on the applications side. The user group changed from the old access group to a new one and the test users suddenly didn't have access to the SuT anymore.

These daily regular load tests will be executed automatically based on a schedule. This execution schedule can be modified according the other workloads which will be necessary when adding other workload patterns to the test execution. As it is enough to execute the exceptional workloads once a release the schedule is modified accordingly. They will be executed as soon as the load test is running stable again after a new deployment.

The tool used for scheduling the workloads is Silk Central [42]. It is from the same vendor as the load generator used which is Silk Performer. Therefore the scheduler is compatible with

the test implementation and workload definition. Once scheduled for each cycle the workload definition and test schedule do not have to be manipulated anymore.

The additional test executions were added during the development phase and after their addition executed with each new release.

5.2.5 Exceptional Workloads

As we now have a longer period of time to execute load and performance tests, we can perform more test runs to conquer certain problems.

WAN Bridge Test. As the application will be used from different global locations, tests where the traffic is sent through the WAN bridge [11] are executed to measure the response times for different locations. Another reason to execute such a test is the importance of the latency as depicted in subsection 3.5.1. The configurations for the latency, packet-loss and bandwidth are taken from the internal measured data for the locations to be tested. The number of concurrent users will be decreased because for a comparison between the locations, there is no need to simulate the whole regular load. Additionally there is a hardware constraint for the load generation through the WAN bridge as we can only use one load agent.

Long Running Regular Load Test. The long running regular load test is primarily used to watch the behaviour of the infrastructure. The runtime of this test is 8 hours and the workload is defined like in the regular load test. The metrics we are interested in are the memory usage and the CPU consumption. This test can validate the sizing of the memory and the CPU with this test run. Additionally this test is also able to observe the behaviour of the memory to identify a possible memory leak. This can only be verified if the heap memory gets full during the test run and the garbage collection is successfully performed without the end user observing any performance degradation or even errors.

Data Input Scalability Test Run. The defined use case is running with adding only one instrument during the workflow. To test the scalability of the SuT an additional regular load test is executed with the addition of three instruments. As the addition of three instruments instead of one affects the subsequent steps in the use case, a whole new test run is needed to test the application for data input scalability. Adding more than three instruments would affect the throughput in a negative way, the addition of more than three instruments for a load test isn't possible. The addition causes the execution of three more clicks and input of two more text fields which causes too much time to reach the desired GST.

5.3 Results

In this section the test results will be presented. Due to failed builds and development issues, the SuT wasn't available during some releases. Therefore some releases could not be tested.

5.3.1 Client-Side Metrics

In this subsection the results of the daily executed tests are briefly discussed. The client-side measured response times for the daily test executions can be found in figure 5.2. The load used is defined in subsection 5.2.4.

The biggest improvement in response time can be found in the 'UC01_02_SelectUniverse' timer between June 6 and June 30. We identified a database query with long execution time through the analysis in the APM. The application team put an index on the respective query which decreased the response time from 10 seconds to a stable 3 seconds. The previously mentioned deployment also decreased the functionality. This can be seen in the stop of the respective measurements. They were included again in the deployment from July 10. New steps were added as new functionality was introduced.

Between July 10 and August 16 there were several incidents that caused the deployment to fail or the test execution to fail. These incidents include:

- Change in the user set up causing our test users to not have access to the system.
- Adding interfaces to subsystems causing the use case to fail in early execution steps.
- Adding new environments in the deployment pipeline which caused the deployments to fail.
- Unavailability of employees due to summer holiday period. This can cause delays on the resolution of any kind of incidents.

Between August 17 and August 18 there are 4 timer which significantly increased. Through the analysis of the respective timer in the APM the identification of a slow database query was possible which was executed during all the steps affected. This finding was reported to the application team. They were able to identify a growth of a database table between the two executions which was caused by a functional tester.

Between August 23 and August 28 a significant change in the GUI caused additional time in implementing the test. Additionally another interface to a subsystem was added causing the use case to fail and not execute the last 5 steps.

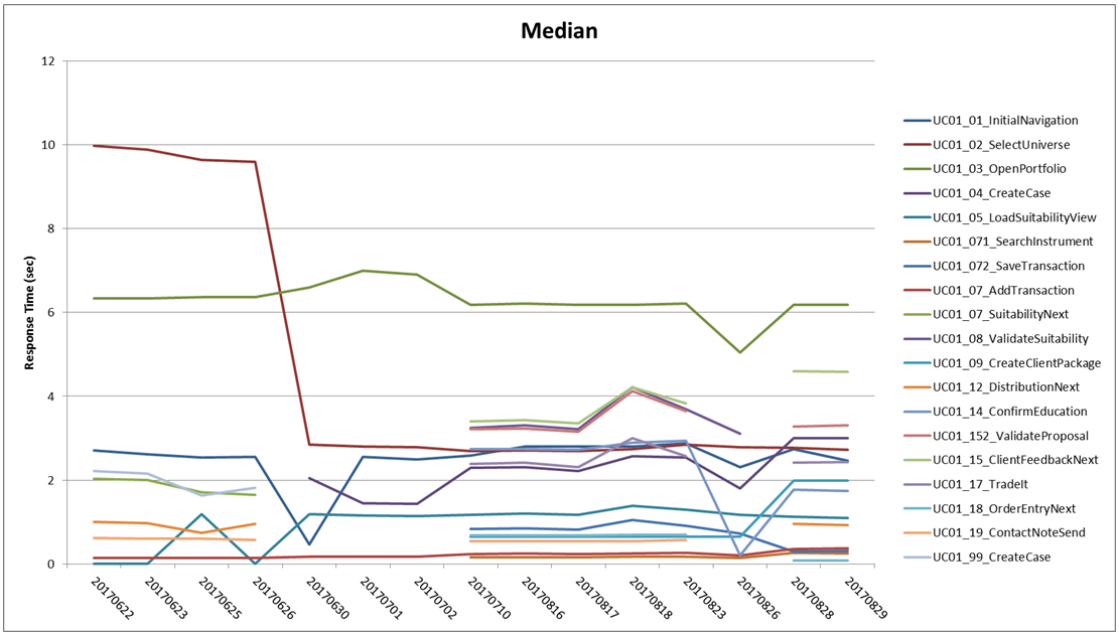

Figure 5.2: Median Response Times for the Daily Regular Load Tests

5.3.2 Server-Side Metrics

In this subsection some exemplary screenshots from the APM are presented and discussed. These are used for illustration purposes and should give the reader a more visible insight in the analysis of the APM results.

Transaction Flow. The analysis in the APM starts by taking a look at the transaction flow in figure 5.3. This gives an overview of the response time contributions of the tiers affected over the tested period of time. This is an example taken from a daily load test defined in subsection 5.2.4. The tiers defined in the subsection 5.1.1 can be found in this transaction flow overview. Additionally the browsers response time contribution is shown as a separate tier. In this case it is also the most time consuming tier shown by the percentage of response time contribution. The amount of inter-tier communication is indicated by the number of calls per minute for example between the application server and the SQL server there are 168.24 per minute. These are calculated for the duration of the load test and in this case defined for a regular load test. As mentioned in subsection 4.5 it is also possible to view the transaction flow of a single transaction/PurePath. In this single PurePath transaction flow the identification of the application server is possible as it is visible in figure 5.4 in the first step. The transactions processing time was 1.1 seconds on the application server which is 48.58 % of the whole response time for the transaction. The next step in the analysis is to look at the execution trace.

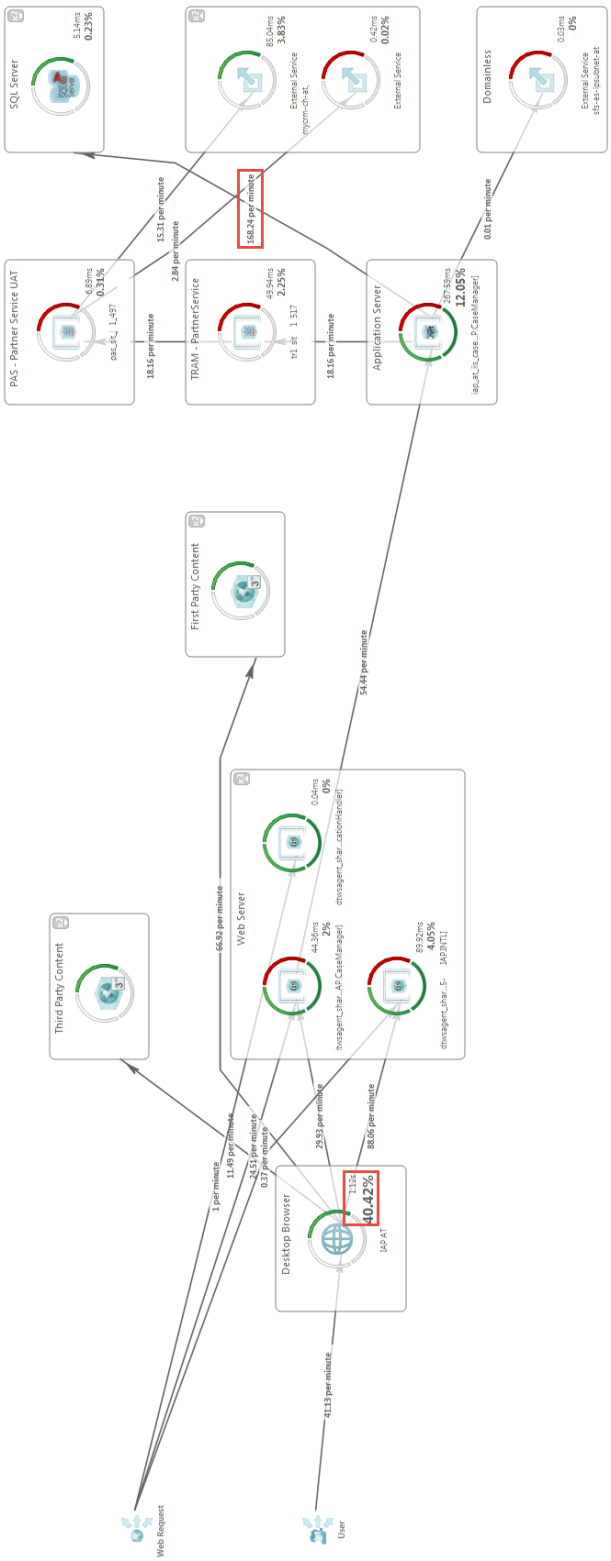

Figure 5.3: Transaction Flow Overview for a Load Test

Execution Trace. The execution trace for a single web request is triggered by a click in the GUI. It is shown in the second and third step of figure 5.4. It is not the complete trace but should give an indication on how the execution times of the methods are visualized. The summarization of the data available is found in subsection 4.5. It is visible that the web request took 1115.14 seconds of execution time. It is also indicated in the second step that there occured an exception. Further breaking down the PurePath gives us the method and the unhandled exception thrown which is shown in step 3. Detailed information on the exception is shown in the last step of the exemplary analysis including the exception stack trace.

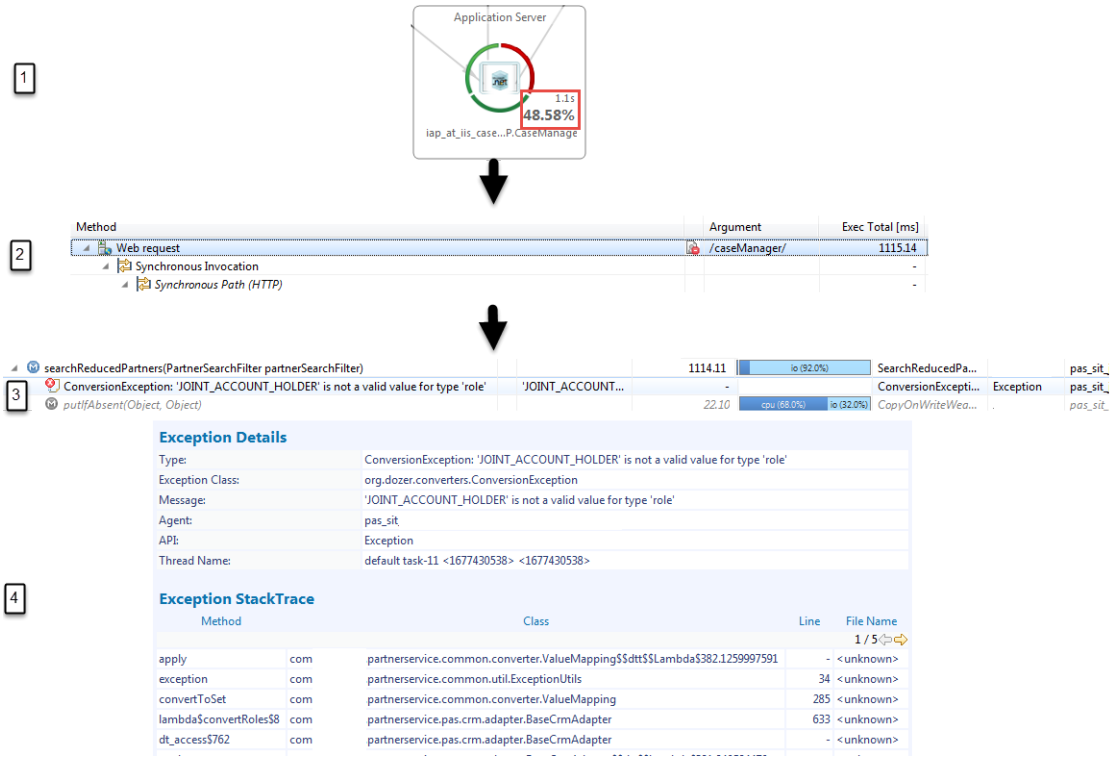

Figure 5.4: Example of Execution Trace Analysis

Data storage. The data storage summary view shows all the executed queries during the load test. An example can be found in figure 5.5. Here the queries are ordered according to the average execution time to identify the longest queries executed during the load test. The APM enables the sorting after all the metrics visible in the screenshot. Important ones are the average execution time and the number of executions per query. It enables the identification of performance critical queries in terms of overall executions during a load test. Drilling down the query with the longest average execution time of 30'088.96 milliseconds leads to the corresponding PurePath/transaction in step 2 of the example. Further drilling down the PurePath the method which queried the SQL on the database is shown with the query in step 3. Additionally it is visible through all the steps of this analysis that an exception occurred on the database. The final step shows additional information to the query. To capture further details to this query the byte-code instrumentation needs to be modified to capture more metrics on the database. This

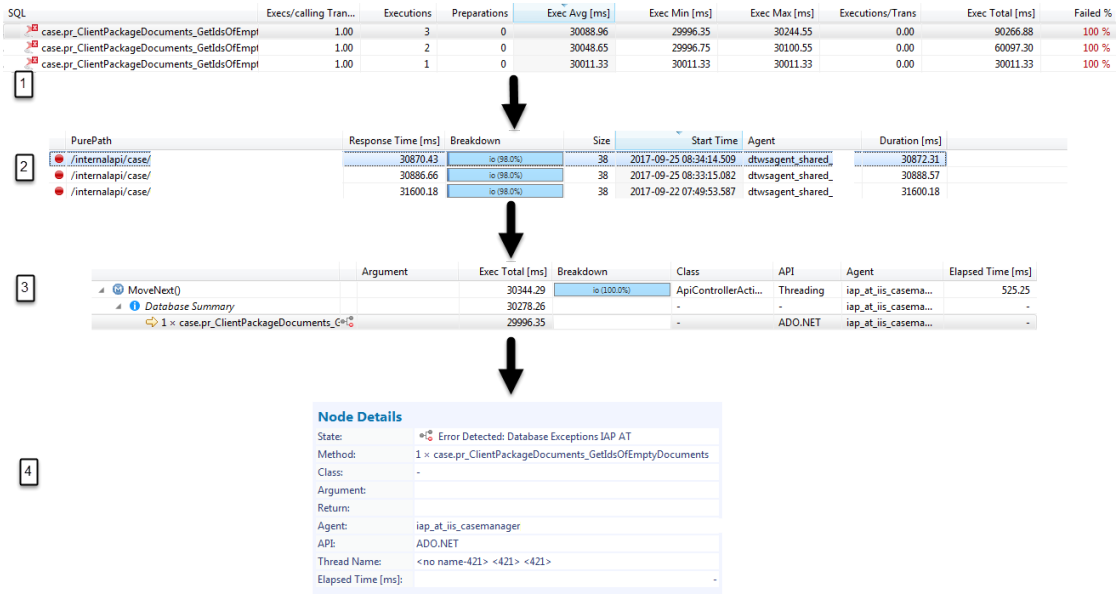

Figure 5.5: Data Storage Example

is a general rule in all the analysis in the APM.

Infrastructure. The APM plots the CPU usage, memory consumption, network utilization and disk space. This is also done for the single processes. In this example the load test execution is shown in the CPU usage as it starts at 0 % utilization and raises up to constant 15 % during the test execution. In this overview sizability problems can be indicated by high CPU usage or high memory consumption. It is only an indicator and needs further clarification if there is a sizing problem. This is dependent on the workload and the planned sizing by the architect. It could also be that in the deployment the configuration has changed. If the heap size got smaller this influences the load test and therefore the change of the memory size can be identified through the load test.

The memory example depicted in figure 5.6 shows the memory consumption. It increases during a load test constantly indicating a possible memory leak. To verify a memory leak is present a long running test execution would be needed which brings the memory to its capacity limit. The behaviour then should be proper garbage collection otherwise a memory leak is identified. In this case a single process was started with the wrong parameters and allocated a heap size which was too small. This is shown in the second step of the example. The heap is filled up and the garbage collection did not clean up the whole heap for seven hours.

5.3.3 Analysis of the WAN-Bridge Benchmark

With the previously introduced WAN-Bridge Test in subsection 5.2.5, a test execution of a local execution and a Singapore execution was conducted. The load was reduced due to test infrastructure constraints, but both location benchmarks were executed with the same load to enable the comparison between the response times. The settings used for Zurich and Singapore were the following:

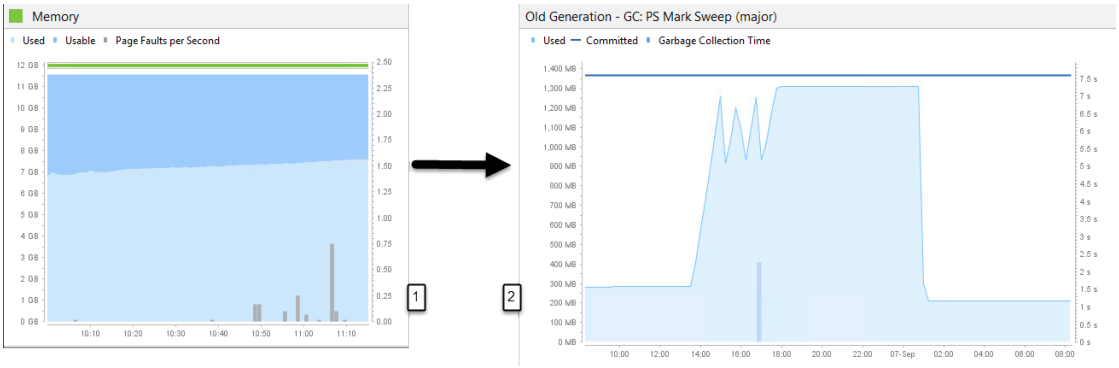

Figure 5.6: Memory Example

	Zurich	Singapore
Latency (ms)	1	173
Bandwidth (MB/sec)	100	100
Packet-loss (%)	0	0.4

Table 5.1: Latency Settings for the WAN-Bridge Test

As we are comparing these two different settings with the identical test setup, it is not necessary to compare the APM monitoring results with each other. The only reason for response time changes are caused by the network settings with the WAN-Bridge. Therefore we analyse the measured end-user response times depicted in figure 5.7.

The 90 % series depict the 90^{th} percentile meaning that 90 % of the measured response times are below that value.

The timers that have the same execution times for the two runs don't include a request to the web server (03_OpenPortfolio, 07_AddTransaction, 071_SearchInstrument and 12_DistributionNext). These user actions only cause client-side processing time in the browser. The other measured response time all include various HTTP requests between the client and the server. The deeper analysis can be found in subsection 5.3.4 in the paragraph GUI Incident.

5.3.4 Incidents identified

Amongst several unhandled exceptions which were easily identified by the exception overview in the APM, some other incidents that required deeper analysis were identified during the tested time period. They are described in the following paragraphs. Also the presented incidents were not solved in the first iteration as the resolution lead to other incidents. An example is the CSS-file incident because the resolution included larger JavaScript files and therefore the performance was not optimized in the first iteration.

GUI Incident. The analysis of the first step in our use case (InitialNavigation) starts with the distinction between client-side incident and server-side incident. To achieve this rough categorization we compare the end-user response time with the processing time in the back-end. For

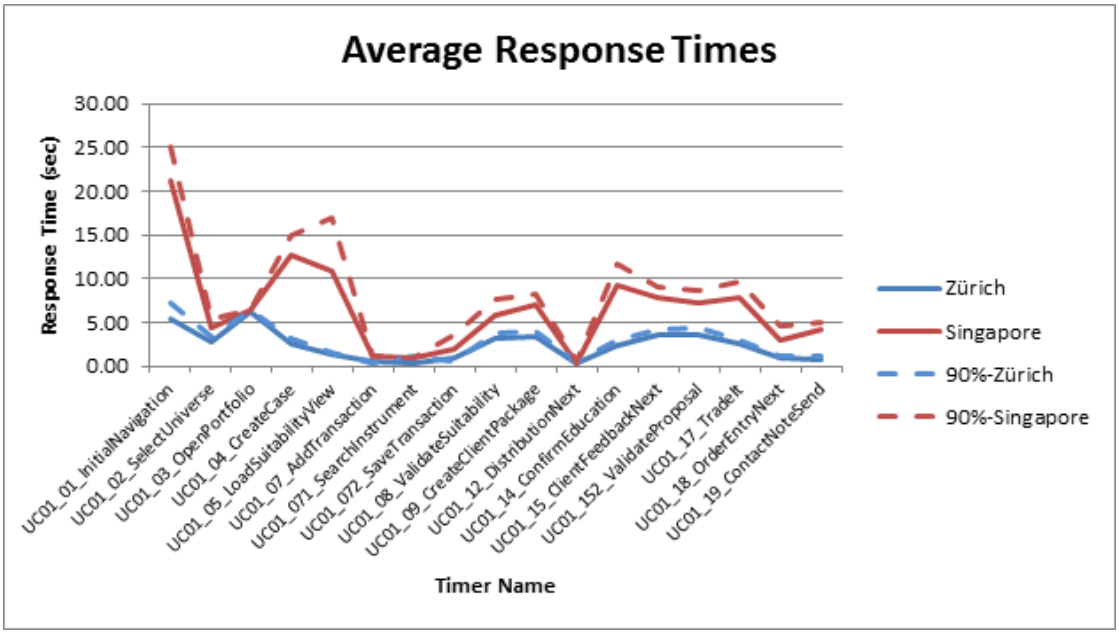

Figure 5.7: Average Response Time Chart WAN-Bridge Test

comparison we use the average response time for one test run, which was 2.4 seconds. The correlating processing time gathered through the APM was roughly 150 milliseconds. This leaves us at the conclusion, that the root cause of this incident is not located on the server side. The next step is the distinction between a client incident or a communication/network incident. In this analysis the usage of the browsers developer tools is helpful. The manual execution of the user action with the Internet Explorers UI Responsiveness function gives a response time contribution overview. The two main contributors in this case are HTML parsing with 1 second and script evaluation with 1 second. These are the main contributors to the measured end-user response time of 2.4 seconds.

Analysis of Observed Errors. During the regular load test execution, the virtual user observed various HTTP 500 errors. These errors caused the failure of 280 use cases out of 756. As HTTP 500 errors are usually caused by a server side root cause, the analysis in the APM is required. During the test execution the APM registered 1120 internal HTTP 500 error responses and 560 on transaction entry. In the respective Pure Paths we are able to identify the cause of the HTTP 500 errors with various occurred exceptions. There were 3 different exceptions which caused the transactions to fail. Two of them involve a subsystem, which wasn't able to deal with the load that was generated. The third exception is a deadlock exception in the database. The related SQL query can also be identified in the APM. It is a query which saves the currently active workflow data in the database. To further illustrate the problem we need to elicit the intended usage of this workflow.

The user can start any suitability proposal and do all the steps required to finish the case immediately. As this is not always possible for the employee to perform the whole suitability proposal, the application automatically saves each modification of the suitability proposal in the database with a delta SQL query. Between the various steps during the workflow this updating

query is the same SQL statement. This is also the query which gets deadlocked 19 times during the regular load test execution. In the APM we can create a Pure Path overview with all the deadlocked transactions. In this overview we are able to identify that there are always two or more Pure Paths started within 1 second, which can be an additional hint in finding the root cause of the deadlock. The next step is the addition of deeper instrumentation in the database to get more insights in this update process. The further analysis isn't completed yet while I'm writing this thesis. It is still a good example in the analysis of observed errors and exceptions during a load test.

Large CSS-File Incident. During the latency test execution the timer 04_CreateCase had the second highest diversity of average response time. The significant change of test setup is the increase of latency from 1 ms to 173 ms. Therefore the analysis starts on the client-server communication. We compare the single user action while observing the client-side network communication. The summarization shows a 1.07 Megabyte CSS-File which is transmitted over the network and therefore influenced heavily by the latency. This net round trip time for 1 ms latency averages at about 125 milliseconds and for 173 ms latency it is over 9 seconds. Therefore the according recommendation is to enable gzip compression on the web server and compress the CSS-File.

Chapter 6

Evaluation

This chapter covers the quantitative evaluation and a brief discussion on the results of the qualitative evaluation.

6.1 Quantitative Evaluation

This section covers the quantitative evaluation of the previous approach and the proposed approach in terms of testing time and an example cost calculation for an occurred error.

6.1.1 Methodology

The methodology chosen for the evaluation of the amount of testing is a simple heuristic. To evaluate the previous approach exemplary data from previous projects is used. Evaluating this approach the case study is used and used for comparison with the previous approach. The error cost calculation for one exemplary incident identified is taken from the NASA technical report [60].

6.1.2 Number of Tested Days and Executions

The number of tested days is strongly dependent on the developing team. If the build, the deployment or interfaces to peripheral systems are failing on the SuT it isn't possible to run any load tests with the approach taken. As a performance tester it is not really possible to influence the build and deployment process. With this approach the tester is completely dependent on this process.

Proposed Approach. An example from the case study is the two-week outage because of a change in the user setup during the testing period which was another unexpected stop to the load test activities. An availability overview of the SuT is given in figure 6.1. In this graph, not available means that the deployment wasn't functionally ready to perform the workflow defined in the use case. During most outages, either the investment dashboard or the case manager wasn't available at all. During the whole 'not available' time of the application there were some test executions, where the first few measurements could be taken, but it wasn't possible to generate the whole load defined.

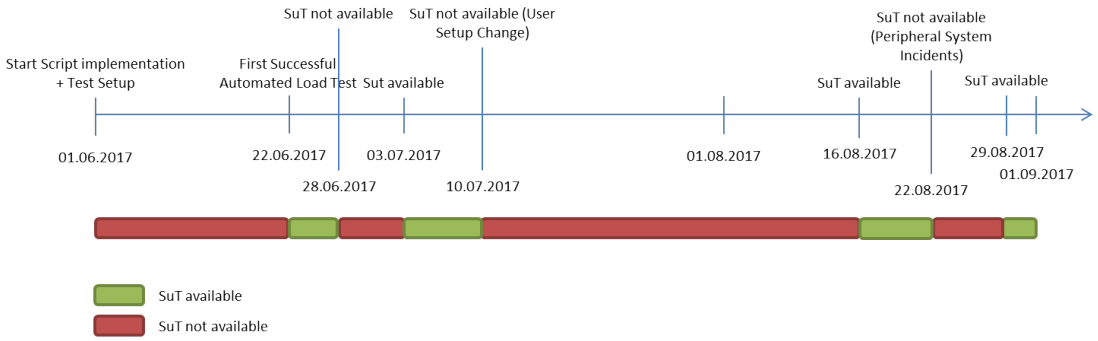

Figure 6.1: Availability of the SuT between June and September 2017

Approach	Previous	Proposed	Difference
# Days of Testing	10	22	12
# Days spent for implementation	2	5	3
# Tested Releases	1	4	3
# Days spent for implementation per Release	2	1.25	0.75
# Days spent for Execution & Reporting	6	11	5
# Days spent for Incident Analysis & Reporting	2	6	4
# Successful Test Runs Executed	6	18	12

Table 6.1: Testing Days Comparison between the Approaches

Previous Approach. To evaluate the number of testing days for the previous approach, we need to make assumptions on the test implementation and setup, as these two metrics differ from the proposed approach. The reason for this is found in the implementation of the test script for the whole workflow in contrary to the evolutionary implementation during the development process. Another metric to be evaluated is the number of days available for testing activities for the previous approach. For this comparison we use the number of UAT testing days from the exemplary release plan described in section 2.4. The time period taken for the development is the same in both cases. In the exemplary release plan we have a trimestrial release which covers 3 months of total time to production. The evaluation from the case study also covers three months. The time spent analysing a load test run will be distinguished between the different types of load test run. In the previous approach the execution of a regular load test, a peak load test and a long running regular load test was executed. The complete time invested for the regular load test run is one and a half days because it is the first test execution and for the peak and long running test run it was one.

In the comparison, depicted in table 6.1, the assumption is taken that the SuT is available for the whole testing period of two weeks. Although this advantageous assumption for the previous approach the number of days of testing is heavily increased in the proposed approach. The days spent for the implementation per release tested are reduced due to the evolutionary development of the test script and due to the maintenance methods presented in section 4.3. The number of test runs is heavily increased due to the daily executed regular load test. However these numbers don't show the addition of the exceptional workloads to the releases tested.

These have evolved through continuously testing and evaluating the problems of the application. The examples can be found in subsection 5.2.5. Especially the addition of the data input scalability test run would have never happened in the previous approach.

The time spent for the test implementation is decreasing for subsequent test runs. For the same deployment the implementation happens only once. For subsequent deployments the implementation time decreases as the script only needs to be updated according to changes or added steps in the workflow. This results in a higher number of test executions for the proposed approach with decreasing effort.

6.1.3 Error Cost Calculation

The error cost escalation with the previous approach and the proposed approach are compared regarding the phase of development where they were detected and fixed in. This is done with the normalized cost-to-fix estimates from the NASA technical report from Stecklein et al. [60]. They summarized different studies of software error cost factors in order to compare it with system cost factors.

For calculation purposes an assumption needs to be taken. In the SCRUM framework the duration of the sprint is the implementation phase. The testing done on the potentially shippable product increment is the testing phase. These phases happen in a two week cycle. Comparing it with the previous development approach depicted in 2.4 these test activities already happen during the development phase. Therefore comparing the error cost factors can be achieved through the comparison of error cost calculations between test and development phase.

The cost-to-fix estimates for the implementation phase, based on the NASA technical report [60], is 10X to 26X and for the test phase it is 50X to 177X. Dividing these numbers by 10 to get a normalized version for the implementation phase results in the software cost factors depicted in table 6.2. The multiplicator is the factor by which the cost is increased. Therefore the cost-to-fix an identified incident is 5 to 6.8 times smaller than in the previous approach. These multiplicators are gained through dividing the cost-to-fix estimates of the testing phase with the development phase.

Cost Comparison. In order to evaluate the cost effectiveness of the proposed approach a few assumptions and numbers are needed. First the internal costs for a performance engineer and a developer are the same. For illustration purposes the further calculation uses a daily salary of 1’500 Swiss francs. The additional costs caused for the proposed approach compared to the previous one are from the performance engineer. In our case study these are additional 12 days of testing which results in 18’000 Swiss francs. The corresponding calculation is as follows:

- C = additional cost
- d = daily salary of a performance engineer
- t = additional days of testing

$$C = d * t$$

Based on internal statistics the cost-to-fix an incident result in 2 days effort in the testing phase. This is a general average to simplify the calculation. Included in these costs are the

	Sofware Cost Factors
Development	1X - 2.6X
Test	5X - 17.7X
Operations	10X - 100X
Multiplicator between Development & Test	5 - 6.8

Table 6.2: Software Cost Factors from the Nasa Technical Report [60]

implementation cost by the developer and cost to build and deploy the fixed version. This results in 3'000 Swiss francs of cost-to-fix for each incident. Based on the previously elicited cost-to-fix multiplicators, this cost can be reduced to 441 to 600 Swiss francs per incident. The resulting savings are 2'400 to 2'559 Swiss francs. In summarization the calculation is as follows:

- s = cost-to-fix savings per incident
- ct = cost-to-fix an incident during testing phase
- m = cost-to-fix multiplicator between testing and development

$$s = ct - (\frac{ct}{m})$$

Dividing the additional costs by the cost-to-fix savings per incident gives us the number of incidents to be identified to equalize the cost:

- i = break-even point of cost = number of incidents to be identified and fixed to equalize cost

$$i = \frac{C}{s}$$

After the identification of 8 incidents over 3 months of testing the proposed continuous approach costs less than the previous approach as $i = \frac{18'000}{2'400} = 7.5$. In our case study the identification and resolution of 13 incidents was achieved during these 3 months. These 13 incidents result in a savings of 31'200 Swiss francs and finally yield in 13'200 Swiss francs less cost.

By inserting the additional cost per day for a performance engineer instead of the complete additional cost we get a general answer. For each day of continuous testing 0.625 incidents need to be identified in order to at least equalize the cost compared to the previous approach. In other words the performance engineer has 1.6 days of continuous testing to identify an incident. This is the reciprocal value to the previous calculation.

The calculation does not take into account the increase in quality of the software. This would increase the value of the proposed approach even more but it is impossible to put up effective numbers for that.

6.2 Discussion

This section covers a brief discussion of the previously performed qualitative evaluation. Additionally the pitfalls of GUI test case implementation is discussed.

6.2.1 Number of Testing Days and Executions

Regarding the total number of days available during the development phase the number of complete test executions is rather disappointing. That is because of the availability of the SuT and the strong dependency on the developers. The complexity of the SuT is also a factor which needs to be considered. The build and deployment process is error-prone in such complex systems with subsystems. Through these subsystems a lot of dependencies for the deployment process are created. If on of the interfaces to subsystems or sub-subsystems fails the SuT is not available for testing.

However in comparison to the previous testing approach the number of test executions and available days for test executions is heavily increased. This is due to the comparably low implementation cost after the initial implementation. The time spent for analyzing the load test results also decreases over time because the performance tester gains more knowledge over time about the system.

6.2.2 Error Cost Calculation

We already know that the earlier an incident is identified the cost to fix it can be drastically reduced. With the proposed approach we were able to identify performance incidents earlier than two weeks before deployment. This helps a project team in categorizing and prioritizing the occurred incident. The proposed approach is especially efficient if a critical error occurs. With the proposed approach the development team has more time to fix the incident before going live with the application. We are able to show that the proposed approach is more cost-efficient than the previous approach after identifying eight or more incidents. However this calculation does not consider the benefits achieved by the increased quality of the software. It is almost impossible to put the increased quality up in effective numbers as this would include an analysis of the end-users perception and saving in time.

6.2.3 Pitfalls of GUI Test Implementation

Implementing performance tests on already completed deployments creates a strong dependency on the successful deployment of the SuT. This is a big pitfall as already shown in the evaluation in subsection 6.1.2. If the SuT is not available due to various reasons the performance engineer can't execute any tests. These reasons are further presented in subsection 5.3.1. Another pitfall is the amount of time spent for the implementation. In the continuous load testing approach the need for updating the test scripts rises after each sprint soaking up time for the performance engineer. Even slightest changes in the GUI like refactoring a button can increase the implementation overhead.

JavaScript executions may also be causes of failing use cases. The test execution happens through a headless browser with the browser API. There are JavaScript executions which can't be reliably triggered through the headless browser. Therefore some test cases will fail certain steps. An example occurred in the case study. When filling a search field the results should dynamically pop up in a window. In the test execution with the headless browser this pop up

didn't show up for about 20 % of the use cases executed. There exist two workarounds for that problem. Either the script retries the step until it is successful which increases the execution time of these transactions. The other possible workaround is triggering the JavaScript event through executing a JavaScript in the browser console during the load test. It is not a completely reliable solution as it also can fail. It is trial and error to see which of these workarounds is more stable. As this implementation pitfall is only shown after a load test it can take few test executions to figure out the best implementation.

Chapter 7

Closing Remarks

In this chapter the research questions from section 1.2 are answered based on the results collected. Further, threats to validity are discussed.

7.1 Research Question 1

The research question covers the possibility of applying the previously used approach for load and performance testing on the browser API. The theoretical answer is delivered in section 2.4 and the evaluation of the practical application in the case study is answered in subsection 6.1.2.

Theoretical Answer. The theory is mainly based on the different development approaches. The previous approach of trimestrial releases did not allow an early implementation of test executions on the GUI through the browser API. With the agile approach of SCRUM it is possible to execute these tests after each cycle because there is a potentially shippable release as a result.

Practical Answer based on the Case Study. In the case study the number of tested days and test executions increased compared to the existing approach with the same development time. This results in the addition of more and different kind of workloads executed. The gain of performance and quality knowledge is also increased due to the large amount of testing. Directly answering the question, it is possible to completely implement the existing approach in an earlier and more agile stage of the software development process. It is even possible to extend it with additional workload definitions and schedule them for future test executions on subsequent releases.

7.2 Research Question 2

The second research question covers the further evaluation of the challenges, advantages and disadvantages compared to the previous approach.

Challenges. The time for testing a release stayed the same compared to the previous approach for our case study example because of the chosen cycle length of two weeks. However this is not a fixed value. For further decreases of release cycle length the main challenge lies in the shorter time period for the whole testing process. This includes reducing the time spent

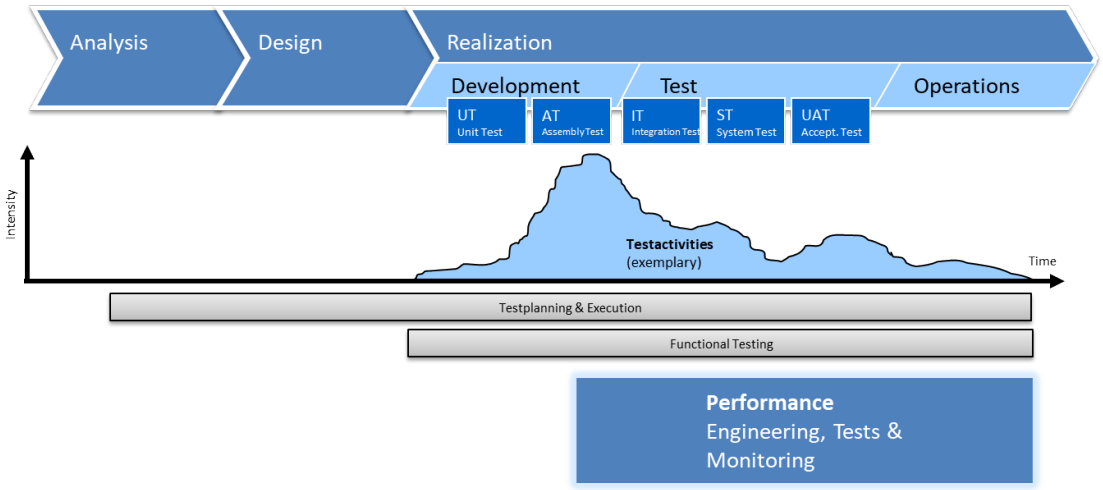

Figure 7.1: Testactivities with this Approach

for the test implementation and the analysis. The test execution time can be reduced by the automatic execution of workloads. Possible improvements in the test implementation are discussed in section 4.3. A rather formal approach of the analysis of performance incidents based on measured slow response times is given in subsection 4.6.

Advantages. The advantages of integrating the tests earlier in the development process are discussed in chapter 6. We can see the number of test executions heavily increased. Also workloads for individual problems could be evaluated and executed for each release as presented in subsection 5.2.5. These could be added to the automatic test execution schedule depicted in subsection 5.2.4. With them being automatically executed for each release the continuous monitoring of certain incidents occurred in the past is possible.

The incidents identified in subsection 5.3.4 can also be dealt with in the subsequent sprint. In our case study example this was done by reengineering the GUI for the large css-file incident. Identifying incidents earlier also helps in prioritizing the incidents properly. In the previous approach there was less time for solving the incidents. The incident solutions with the previous approach often included the implementation of workarounds instead of solving the root cause of the incident.

Another advantage is the gain of knowledge by the performance tester. As mentioned in section 2.4 the performance engineer is in a separate team and not involved in the development process of a project. Therefore the first phase is getting to know the SuT. If this gained knowledge can be used continuously, as in the agile approach, it is more efficient than when it is used for only testing one release.

The test activities from the previous approach depicted in section 2.4 included a shift-left in the proposed approach which is shown in figure 7.1.

Disadvantages. The time, and therefore money, spent for testing is increased heavily. It is hard to compare the use of the approach taken in this thesis to the previous approach. Pondering if it is worth investing more time for performance testing in such an early phase where a lot of changes are still happening is difficult. It definitely increases the quality of the soft-

ware. Evaluating the efficiency between the existing and the proposed approach is impossible to achieve.

An additional disadvantage is the close involvement in the project. The tester gets an in-depth view of the project through the continuous involvement during the development process. This can lead to project blindness with the loss of the outside view. Especially for negative test results, the neutral opinion can be influenced due to the sympathy to the project team which could cause falsified test reports. Another example is the

7.3 Threats to Validity

In the following section threats to validity of this thesis are further elicited.

Only one Example. The evaluation is based on only one exemplary case study. It is not a general evaluation and prove that this approach is working efficiently. The example taken is done for a fairly small test suite with only one use case. For larger test suites with more use cases the time spent for implementing and analysing can be completely different. Logically, for more test cases the time and effort spent for the implementation is be increased. It is not evaluated if this effort increases linearly or exponentially. The same counts for the analysis of a test run.

Focus on the Financial Institution. Only focusing on the load and performance testing for financial applications with a predefined workflow is a special case and not a general solution. The use case definition depicted in subsection 4.2.1 is simplified by that fact. Also the workload definition and calculation depicted in 4.2.4 is simplified. They don't have to be evaluated continuously as it would be the case for publicly accessible applications with a high variance in usage.

Performance Test Suite. The usage of the performance test suite from Microfocus (Silk Performer [43] and Silk Central [42]) for the test execution, implementation and scheduling is a simplification which is not open source and therefore not accessible to the public. It simplified all the steps involved. For example the pacing with a GST (depicted in subsection 4.2.4) for the test case would take more time with any other tool used. This would increase the implementation and execution time. Also the long term scheduling for the load tests is fairly easy with the support of Silk Central.

Assumptions The assumptions made in the evaluation can be criticized. Especially that IT professionals have the same salary is not realistic but simplifies the calculation. Otherwise all involved parties in solving an incident would have to have single salaries as resolving an error does not only include effort from the developer. It also includes deployment effort from a separate responsible or involvement from a database responsible. Also the assumption on the error resolution time can vary significantly depending on the error. For calculation purposes an average value has been chosen.

Chapter 8

Glossary

Application Performance Management (APM). This is a monitoring solution, which will be integrated in the environment to be tested. It will measure key performance metrics.

Benchmark. In software testing the term benchmark is usually used to evaluate the relative performance of hardware components. However the oxford dictionary defines the term benchmark as follows: "A standard or point of reference against which things may be compared." [20]. Therefore in this thesis the term benchmark is used to assess the relative performance of different geographic locations in subsection 5.3.3 or different releases in 5.2.4.

Environment. An instantiation (deployment) of the application to be tested. In a software development process, there should be multiple environments dedicated to testing purposes.

Key Performance Metrics (KPIs). These are metrics used to determine the performance of the application. These metrics include Memory consumption, CPU usage and response times of either end-user actions, single method executions or SQL queries on the database. This also includes the network impact on the performance (Network Utilization). KPI's are defined for specific workloads and therefore can differ for different kinds of load test runs.

Load generator. The tool to be used, to generate load on the environment. This can be done through triggering unit test cases, functional test cases or end-user use case simulation on the application to be tested.

Service Level Agreement (SLA). Definition: "A service level agreement (SLA) defines the idea of a reliable contract between servic providers an their users. It contains the information about the responsibilities, the scope, and the expected quality of service (QoS)." Walter et al. [70]. Additionally it includes the definition of KPIs for certain load pattern.

Test Run. A Test Run is an execution of a workload on the environment to be tested.

Thinktime. The thinktime refers to the time a user needs until he interacts with the application. It can be time the user needs to read certain passages or find a button in the GUI. It doesn't include page load times or insertions of text.

Transaction. A transaction is one execution of a test case. In a performance test, these transactions will be executed multiple times in order to get significant results on the response times. To determine an applications performance, it is not enough the execute just one transaction and measure the elapsed time because it doesn't deliver statistically significant results on the applications performance. To define a load, the measure transactions per hour is used.

User Acceptance Test (UAT). In the development lifecycle, this is the last stage of testing before the deployment to production of a new release. It covers end-user use case execution testing, including performance testing.

Workload. These are the definitions of the load patterns, which will be executed during a single test run. In the case of executing unit tests or functional test cases, this contains the time of the test execution and the number of executions of the respective tests during the defined time period. In end-user use case simulations, there is additionally the definition of the number of users to be used during the test period. The workload also specifies the type of the performance test, which will have a great impact on the sizing of the previously defined numbers.

Bibliography

[1] T. M. Ahmed, C.-P. Bezemer, T.-H. Chen, A. E. Hassan, and W. Shang. Studying the effectiveness of application performance management (apm) tools for detecting performance regressions for web applications: An experience report. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 1–12, New York, NY, USA, 2016. ACM.

[2] W. Ahn, J. Choi, T. Shull, M. J. Garzarán, and J. Torrellas. Improving javascript performance by deconstructing the type system. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 496–507, New York, NY, USA, 2014. ACM.

[3] Akamai. The impact of web performance on e-retail success. *Akamai White Paper*, 2004.

[4] Akamai. `https://www.akamai.com/us/en/about/news/press/2006-press/akamai-and-jupiterresearch-identify-4-seconds-as-the-new-threshold-of-acceptability.jsp`, Nov 2006.

[5] T. Angerstein, D. Okanović, C. Heger, A. van Hoorn, A. Kovačević, and T. Kluge. Many flies in one swat: Automated categorization of performance problem diagnosis results. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, pages 341–344, New York, NY, USA, 2017. ACM.

[6] AppDynamics. `https://www.appdynamics.com/`. Accessed: 2017-07-21.

[7] W. Bays and K.-D. Lange. Spec: Driving better benchmarks. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 249–250, New York, NY, USA, 2012. ACM.

[8] S. Becker, H. Koziolek, and R. Reussner. Model-based performance prediction with the palladio component model. In *Proceedings of the 6th International Workshop on Software and Performance*, WOSP '07, pages 54–65, New York, NY, USA, 2007. ACM.

[9] P. C. Brebner. Real-world performance modelling of enterprise service oriented architectures: delivering business value with complexity and constraints. *SIGSOFT Softw. Eng. Notes*, 36(5):85–96, Sept. 2011.

[10] P. C. Brebner. Automatic performance modelling from application performance management (apm) data: An experience report. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ICPE '16, pages 55–61, New York, NY, USA, 2016. ACM.

[11] W. Bridge. https://code.google.com/archive/p/wanbridge/.

[12] J. Brutlag. http://services.google.com/fh/files/blogs/google_delayexp.pdf, Jun 2009.

[13] R. Callan, F. Behrang, A. Zajic, M. Prvulovic, and A. Orso. Zero-overhead profiling via em emanations. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 401–412, New York, NY, USA, 2016. ACM.

[14] L. Cheng, S. Kotoulas, T. E. Ward, and G. Theodoropoulos. High throughput indexing for large-scale semantic web data. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 416–422, New York, NY, USA, 2015. ACM.

[15] J. Cito, G. Mazlami, and P. Leitner. Temperf: Temporal correlation between performance metrics and source code. In *Proceedings of the 2Nd International Workshop on Quality-Aware DevOps*, QUDOS 2016, pages 46–47, New York, NY, USA, 2016. ACM.

[16] V. Cortellessa, A. Di Marco, R. Eramo, A. Pierantonio, and C. Trubiani. Digging into uml models to remove performance antipatterns. In *Proceedings of the 2010 ICSE Workshop on Quantitative Stochastic Models in the Verification and Design of Software Systems*, QUOVADIS '10, pages 9–16, New York, NY, USA, 2010. ACM.

[17] V. Cortellessa, A. D. Marco, and C. Trubiani. Performance antipatterns as logical predicates. In *Proceedings of the 2010 15th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS '10, pages 146–156, Washington, DC, USA, 2010. IEEE Computer Society.

[18] M. Czerwinski, E. Horvitz, and S. Wilhite. A diary study of task switching and interruptions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, pages 175–182, New York, NY, USA, 2004. ACM.

[19] M.-W. O. Dictionary. https://www.merriam-webster.com/dictionary/science. Accessed: 2017-08-08.

[20] O. Dictionary. https://en.oxforddictionaries.com/definition/benchmark. Accessed: 2017-09-07.

[21] DynaTrace. https://www.dynatrace.com/. Accessed: 2017-07-21.

[22] DynaTrace. https://community.dynatrace.com/community/display/DOCDT65/Bytecode+Injection. Accessed: 2017-08-21.

[23] M. Ermuth and M. Pradel. Monkey see, monkey do: Effective generation of gui tests with inferred macro events. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2016.

[24] V. Ferme and C. Pautasso. Towards holistic continuous software performance assessment. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ICPE '17 Companion, pages 159–164, New York, NY, USA, 2017. ACM.

[25] E. Foundation. https://www.eclipse.org/. Accessed: 2017-09-27.

[26] E. Fullerton. Akamai technologies - 2014 consumer web performance expectations survey. *Akamai Technologies - 2014 Consumer Web Performance Expectations Survey*, 2014.

[27] Gartner. http://www.gartner.com/technology/research/methodologies/magicQuadrants.jsp. Accessed: 2017-08-08.

[28] Gartner. https://www.gartner.com/doc/reprints?id=1-3M8KIVD&ct=161118&st=sb. Accessed: 2017-09-08.

[29] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM.

[30] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. Technical report, Department of Electronics and Information Systems, Ghent University, Belgium, 2007.

[31] Google. https://developers.google.com/speed/pagespeed/. Accessed: 2017-09-18.

[32] M. Grechanik, C. Fu, and Q. Xie. Automatically finding performance problems with feedback-directed learning software testing. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 156–166, Piscataway, NJ, USA, 2012. IEEE Press.

[33] C. Heger, J. Happe, and R. Farahbod. Automated root cause isolation of performance regressions during software development. Technical report, Karlsruhe Institute of Technology, Karlsruhe, Germany, 2013.

[34] C. Heger, A. van Hoorn, M. Mann, and D. Okanović. Application performance management: State of the art and challenges for the future. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, pages 429–432, New York, NY, USA, 2017. ACM.

[35] C. Heger, A. van Hoorn, D. Okanovic, S. Siegl, and A. Wert. Expert-guided automatic diagnosis of performance problems in enterprise applications. In *12th European Dependable Computing Conference, EDCC 2016, Gothenburg, Sweden, September 5-9, 2016*, pages 185–188, 2016.

[36] R. Heinrich, A. van Hoorn, H. Knoche, F. Li, L. E. Lwakatare, C. Pahl, S. Schulte, and J. Wettinger. Performance engineering for microservices: Research challenges and directions. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ICPE '17 Companion, pages 223–226, New York, NY, USA, 2017. ACM.

[37] Z. M. Jiang. Automated analysis of load testing results. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 143–146, New York, NY, USA, 2010. ACM.

[38] T. Kalibera, J. Lehotsky, D. Majda, B. Repcek, M. Tomcanyi, A. Tomecek, P. Tuma, and J. Urban. Automated benchmarking and analysis tool. Technical report, Charles University, Prague, Czech Republic, 2006.

[39] Kieker. http://kieker-monitoring.net/. Accessed: 2017-07-21.

[40] G. Linden. http://glinden.blogspot.ch/2006/11/marissa-mayer-at-web-20.html, Nov 2006.

[41] M. Mehrara and S. Mahlke. Dynamically accelerating client-side web applications through decoupled execution. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 74–84, Washington, DC, USA, 2011. IEEE Computer Society.

[42] Microfocus. `https://www.microfocus.com/de-de/products/silk-portfolio/silk-central/`. Accessed: 2017-09-14.

[43] Microfocus. `https://www.microfocus.com/de-de/products/silk-portfolio/silk-performer/`. Accessed: 2017-09-19.

[44] Y. Na, S. W. Kim, and Y. Han. Javascript parallelizing compiler for exploiting parallelism from data-parallel html5 applications. *ACM Trans. Archit. Code Optim.*, 12(4):64:1–64:25, Jan. 2016.

[45] A. Nederlof, A. Mesbah, and A. v. Deursen. Software engineering for the web: The state of the practice. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 4–13, New York, NY, USA, 2014. ACM.

[46] C. Nicoll. `https://www.scrum.org/forum/scrum-forum/5625/experience-report-move-waterfall-scrum`. Accessed: 2017-09-20.

[47] J. Nielsen. `ttps://www.nngroup.com/articles/website-response-times/`, June 2010.

[48] D. Okanovic, A. van Hoorn, C. Heger, A. Wert, and S. Siegl. Towards performance tooling interoperability: An open format for representing execution traces. In *Computer Performance Engineering - 13th European Workshop, EPEW 2016, Chios, Greece, October 5-7, 2016, Proceedings*, pages 94–108, 2016.

[49] P.-S. PCM. `http://www.palladio-simulator.com/`. Accessed: 2017-07-25.

[50] M. Peiris and J. H. Hill. Towards detecting software performance anti-patterns using classification techniques. *SIGSOFT Softw. Eng. Notes*, 39:1–4, Feb. 2014.

[51] Phabricator. `https://secure.phabricator.com/book/phabcontrib/article/n_plus_one/`. Accessed: 2017-09-22.

[52] M. Pradel, P. Schuh, G. C. Necula, and K. Sen. Eventbreak: analyzing the responsiveness of user interfaces through performance-guided test generation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 33–47, 2014.

[53] R. Ramakrishnan, V. Shrawan, and P. Singh. Setting realistic think times in performance testing: A practitioner's approach. In *Proceedings of the 10th Innovations in Software Engineering Conference*, ISEC '17, pages 157–164, New York, NY, USA, 2017. ACM.

[54] N. Relic. `https://newrelic.com/`. Accessed: 2017-07-21.

[55] A. Rodge, S. K. Soni, L. Chinnaga, P. Johari, J. Bose, C. Pramanik, and A. Bhide. Scalable and optimal load generation for aws clients. In *Proceedings of the 8th Annual ACM India Conference*, Compute '15, pages 95–100, New York, NY, USA, 2015. ACM.

[56] Scrum.org. https://www.scrum.org/resources/scrum-framework-poster. Accessed: 2017-09-07.

[57] M. Sedaghat, F. Hernandez-Rodriguez, and E. Elmroth. A virtual machine re-packing approach to the horizontal vs. vertical elasticity trade-off for cloud autoscaling. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, CAC '13, pages 6:1–6:10, New York, NY, USA, 2013. ACM.

[58] SLA. http://www.service-level-agreement.net/. Accessed: 2017-09-26.

[59] S. Souders. https://blog.mozilla.org/metrics/2010/03/31/firefox-page-load-speed-part-i/, Mar 2010.

[60] J. M. Stecklein, J. Dabney, B. Dick, B. Haskins, R. Lovell, and G. Moroney. Error cost escalation through the project life cycle. Conference paper, NASA Johnson Space Center; Houston, TX, United States, 2004.

[61] C. Stewart and K. Shen. Performance modeling and system management for multi-component online services. Technical report, Department of Computer Science, 2005.

[62] S. Sundaresan, N. Magharei, N. Feamster, R. Teixeira, and S. Crawford. Web performance bottlenecks in broadband access networks. *SIGMETRICS Perform. Eval. Rev.*, 41(1):383–384, June 2013.

[63] M. D. Syer, Z. M. Jiang, M. Nagappan, A. E. Hassan, M. Nasser, and P. Flora. Continuous validation of load test suites. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, pages 259–270, New York, NY, USA, 2014. ACM.

[64] E. Thereska, B. Doebel, A. X. Zheng, and P. Nobel. Practical performance models for complex, popular applications. Technical report, Microsoft Research, Cambridge, United Kingdom, 2010.

[65] C. Trubiani and A. Koziolek. Detection and solution of software performance antipatterns in palladio architectural models. In *Proceedings of the 2Nd ACM/SPEC International Conference on Performance Engineering*, ICPE '11, pages 19–30, New York, NY, USA, 2011. ACM.

[66] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1041–1052, New York, NY, USA, 2017. ACM.

[67] G. Vergori, D. A. Tamburri, D. Perez-Palacin, and R. Mirandola. Devops performance engineering: A quasi-ethnographical study. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ICPE '17 Companion, pages 127–132, New York, NY, USA, 2017. ACM.

[68] I. Švogor. An initial performance review of software components for a heterogeneous computing platform. In *Proceedings of the 2015 European Conference on Software Architecture Workshops*, ECSAW '15, pages 16:1–16:4, New York, NY, USA, 2015. ACM.

[69] J. Waller, N. C. Ehmke, and W. Hasselbring. Including performance benchmarks into continuous integratino to enable devops. Technical report, Kiel University, Kiel, Germany, 2015.

[70] J. Walter, D. Okanović, and S. Kounev. Mapping of service level objectives to performance queries. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ICPE '17 Companion, pages 197–202, New York, NY, USA, 2017. ACM.

[71] J. Walter, A. van Hoorn, H. Koziolek, D. Okanovic, and S. Kounev. Asking "what"?, automating the "how"?: The vision of declarative performance engineering. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ICPE '16, pages 91–94, New York, NY, USA, 2016. ACM.

[72] C. Weiss, D. Westermann, C. Heger, and M. Moser. Systematic performance evaluation based on tailored benchmark applications. Technical report, SAP Research, Karlsruhe, Germany, 2013.

[73] A. Wert, J. Happe, and L. Happe. Supporting swift reaction: Automatically uncovering performance problems by systematic experiments. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 552–561, Piscataway, NJ, USA, 2013. IEEE Press.

[74] F. Willnecker, A. Brunnert, W. Gottesheim, and H. Krcmar. Using dynatrace monitoring data for generating performance models of java ee applications. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, pages 103–104, New York, NY, USA, 2015. ACM.

[75] M. Woodside, G. Franks, and D. C. Petriu. The future of software performance engineering. In *2007 Future of Software Engineering*, FOSE '07, pages 171–187, Washington, DC, USA, 2007. IEEE Computer Society.

[76] X. Xiao, S. Han, D. Zhang, and T. Xie. Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 90–100, New York, NY, USA, 2013. ACM.

[77] Yahoo. `http://yslow.org/`. Accessed: 2017-09-18.

[78] E. Zimran and D. Butchart. Performance engineering throughout the product life cycle. CompEuro '93. 'Computers in Design, Manufacturing, and Production', 1993.

[79] M. Züger, C. Corley, A. N. Meyer, B. Li, T. Fritz, D. Shepherd, V. Augustine, P. Francis, N. Kraft, and W. Snipes. Reducing interruptions at work: A large-scale field study of flowlight. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, pages 61–72, New York, NY, USA, 2017. ACM.