TALLINN UNIVERSITY OF TEHNOLOGY
Faculty of Information technology
Department of Computer Science

A framework for empirical evaluation of Java application performance

Master's thesis

Student: Viktor Reinok
Student code: 133415IAPM
Supervisor: PhD Juhan-Peep Ernits

Tallinn 2016

Author’s declaration

Herewith I declare that this thesis is based on my own work. All ideas, major views and data from different sources by other authors are used only with a reference to the source. The thesis has not been submitted for any degree or examination in any other university.

(*signature*)

(*date*)

Raamistik Java rakenduse jõudluse probleemide lahendamiseks empiirilise lähenemisega

Annotatsioon

Selles töös tutvustatakse raamistikku, millega saab tuvastada ja jälgida Java rakenduse jõudluse probleeme arendus tsüklis. Raamistik on ettenähtud osana pidevkooste protsessist, et hinnata järjepidevalt rakenduse muudatuste mõju jõudlusele. See raamistik pakub tagasisidet arendajale informatsiooniga võimalikest jõudluse probleemidest ja probleeme kirjeldavaid kvantitatiivseid väärtusi, analüüsimaks probleemi relevantsust.

Töö raames tehti tehtud uuring turul olemasolevatest jõudluse seire vahenditest, tehnoloogiast, millel põhinevad uuritavad vahendid ja seotud akadeemilistest töödest. Selle uuringu põhjal valitud välja rakendusega ühilduv tööriist, et koguda andmeid, mida saab kasutada jõudluse probleemide tuvastamiseks. Lisaks loodi koormust genereeriv API ja andmeid analüüsiv tööriist, mis realiseeriti Javas. Koormuse generaatorit kasutati selleks, et tekitada juhitavat koormust uuritavale rakendusele ja analüüsivat tööriista selleks, et uurida koormuse mõju ja teha kvantitatiivseid järeldusi, mida omakorda kasutajale kuvada.

Valideerimaks raamistiku tõhusust sai läbiviidud kaks uuringut. Raamistikku sai kasutati nii väiksema veebiraamistiku näidisrakenduse peal, kui ka suuremas reaalselt kasutuses oleva rakenduse peal. Mõlemad uuringud korral aitas käesolevas töös loodud raamistik tuvastada jõudlusprobleeme. Lisaks, avastati suuremast rakendusest ka varem mitteteadaolevaid jõudlusprobleeme.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 63 leheküljel, 6 peatükki, 14 joonist, 5 tabelit ja üle 4000 rea koodi avalikult kättesaadavas Bitbucket-i koodi hoidlas [1]

A framework for empirical evaluation of Java application performance

Abstract

A framework for detecting and tracking performance problems for Java applications is developed in the thesis. The framework is intended to be a step in the continuous integration process to evaluate effect of progressive application modifications on performance. It provides feedback for a developer and points out possible performance problem type and a quantitative measure to analyze its severity.

Research was done to provide an overview of application performance monitoring tool market, technology used by the tools and related academic work. Based on the research a tool was chosen for probing the application to collect data which could be used to detect performance issues. In addition to probing tool and load generation API and analyzer tool was implemented in Java to induce generation and process data from the application.

To validate the effectiveness of the framework two case studies were made. Framework was used with a smaller and a large enterprise scale application. Both case studies showed proofed that the approach provides practical value and the framework also support large applications. In the latter case it was possible to discover performance issues not known previously.

The thesis is in English and contains 63 pages of text, 6 chapters, 14 figures, 5 tables and over 4000 line of code in public Bitbucket repository. [1]

Glossary of terms, acronyms and abbreviations

BCI	***Bytecode instrumentation*** Bytecode-i instrumenteerimine
APM	***Application performance monitoring*** Rakenduse jõudluse monitooring
AIO	***All-In-One*** *– A large application used for a case study* *Rakendus mida kasutakse juhu uuriguks*
Instrumentation	***In current context and in Java world generally instrumentation means manipulation of bytecode for gathering data or modifying executable code behavior*** Antud kontekstis ja üleültdse Java see tähendab byte-koodi manipuleerimist andmete kogumise ja käivitatava koodi käitumise muutmist
REST	***Representational state transfer*** Esitlus oleku ülekanne
SOAP	***Simple object access protocol*** Lihtsa objekti ligipääsu protokoll
JVM	***Java Virtual Machine*** Java virtuaalne masin
CI	***Continuous integration*** *Järjepidev juurutamine*
JVMTI	***JVM tool interface is a programming interface used by development and monitoring tools*** Java virtuaal masina tööriista liides
CCT	***Calling context tree*** Programmi täitmise konteksi puu

Figures

Tables

Table of contents

1. Introduction

Performance is one of the many characteristics of any software system. Often with the growth of a system and increase in complexity, the importance of the performance aspects increases mainly due to becoming a limiting factor for future growth of the system. During the application exploitation and maintenance the complexity of the system usually grows and tracking or even narrowing down the root cause of a problem is very difficult.

Serious performance issues often occur under higher than normal load of an application. Such load can often cause highly concurrent use of some resources which may become a bottleneck in the normal workflow of an application.

During the development of software, that already has a history of being in production there already exists some baseline for performance related nonfunctional requirements. As the incremental changes are made, the overall complexity of the system increases, and that can often have with negative effects on performance. But typically there is no clear way to give feedback to the developer who made the incremental changes.

In addition during the lifetime of a software system even with an unchanged initial request throughput requirement, the complexity of the software system tends to increase. Maintenance, especially rapid requirement changes and feature requests are causing degradation of application’s throughput in nontrivial ways

To counter the problems described above we propose an approach to track down future performance issues and integrate the appropriate regression of performance related to nonfunctional requirements into the continuous integration (CI) loop. The tool framework will provide the means to run experiments in a controlled environment, where the cause of most relevant performance problems will get narrowed down to improve the quality of an application.

1.1 Problem and Background

Most frequently used software development methods focus mainly on agility and the functional correctness of software while the performance of the application including adhering to performance related nonfunctional requirements, remains in the background. It is a known fact that fixing performance problems in the late stages of the lifecycle of the application may require considerable adjustments in the design, for example at the architectural level.

Performance matters because higher than normal response time causes end-user abandonment rate to grow almost exponentially [2] and the conversion rate goes down [3] without mentioning of end-user satisfaction. Thus it is in the interest of the business to keep the response times of the underlying application normal.

Based on my professional experience, it can be said that performance metrics do not tend to get enough priority in the development process. By introducing an approach that enables to detect effects of changes in the long term, e.g. validate the application in the presence of significantly larger amounts of customers or data, we will increase the sustainability and maintainability of a software system.

A proven way to run tests and track their success or failure is continuous integration (CI). In CI, tests get run periodically and the effect of each change is measured. An example of such a build automation tool is Jenkins [4]. There already exist various plugins for Jenkins unit-test based regression that already outputs a list of failed tests and points which commit to a software configuration management triggers such failures. Similar functionality could be used for analyzing performance issues and validating performance fixes. [5]

As for an application in mature lifecycle stage there is already a baseline for each component of the application so if there are some performance issues they are reported as individual cases. But still major issues come up in case of seldom occurring abnormal system load which affects individual software components which are not evenly scaled to handle the load. To test the application for such issues load generation testing combined with detailed performance monitoring could be used.

The thesis provides a framework for Java to adapt performance related metrics based on current baseline which is especially useful in the later stages of the application’s life cycle. The framework consist of three main components. The load generator is customizable for

specific application, a probe which will collect relevant application execution data and an analysis module which will present the data in the most relevant way.

1.2 Goal setting

Main goal of this thesis is to build an application framework for measurement and tracking of performance issues in Java applications. The approach is targeted to aid the tracking and solving of performance problems for applications at mature stages of the life cycle. The main goal is divided into the following sub goals:

- To choose the most suitable probing solution which is attachable to a deployed system in the testing environment. To gather data which is most helpful for a developer, to analyze the root cause of some performance issues. Data such as execution time of different application layers and mainly calling context traces with time measurement of time spent in each component. An example of the calling context tree is given in Figure 1:

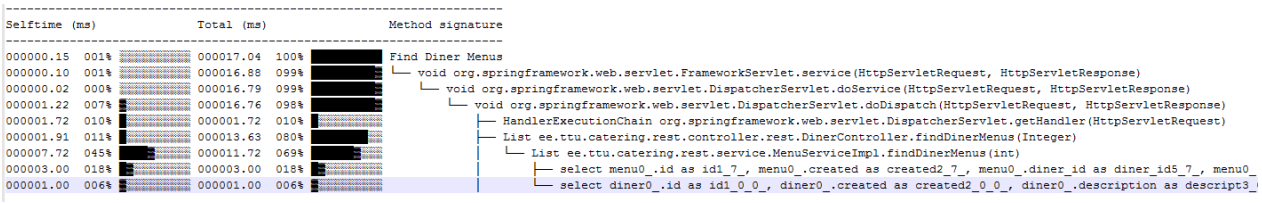

Figure 1 Calling context tree

- Create a customizable load generation API for generic Java web application and provide a specific implementation for existing large scale application.

- Create an analytical module which compares multiple data sets recorded by the probing solution to analyze aggregated metrics such as software component execution time inside calling context trace. Analysis should highlight software components which are progressively increasing response time and decreasing throughput.

1.3 Methodology

To choose the most suitable probing solution to collect data from the application which could be used to detect performance problems, a research have to be done mainly in two directions. First we carry out a broad review of most used application performance monitoring tools. The review will diminish the probability of implementing something which is already there and provide informative overview of performance monitoring tools available. Secondly, a review about related academic work that is based on the technology used by application monitoring tools. Also a detailed overview of the underlying technology should be provided which will aid to understand performance tools.

For creating a custom load generation tool which is a crucial component of the framework conventional programming methods could be used. An implementation of a customer tool is necessary because of using tool such as JUnit [6] require too much application specific configuration.

To create analytical tool a smaller example application should be used with known performance problem to validate effectiveness of the framework. Also there should be two revision of the smaller example application. One with the known performance problem and another with the fix to validate empiric aspect of the framework. Also the framework have to be tested with large application.

1.4 Outline of the Thesis

In the following chapter 2, a review of application monitoring tools available on the market is presented. In addition to the market review in section 2.1 an overview of academic work on the topic of underlying technology (bytecode instrumentation) used in application monitoring tools. Based on research a profiling component of the framework is chosen.

Next chapter number 3 goes in depth to bytecode instrumentation. Minimalistic descriptive code examples of Java support and most used bytecode instrumentation APIs are presented.

Further, chapter 4 continuous with describing implemented framework and each component. A description of the process is provided in Section 4.2 with a comparison to conventional load testing is section 4.3. The architecture of the framework is described in section 4.4.

Finally the case studies of using the framework are presented in chapter 5. Results of the approach are provided for a larger application case study in section 5.2.2 and smaller in section 5.1.1.

2. Related work and available application performance monitoring tools

The Java ecosystem is well known for its strong side - tooling. There are several tools which are included in the JDK. JConsole and VisualVM can be used for profiling JVM processes and providing a good overview of an impact on resource consumption and memory usage metrics.

On the other hand there are exist several useful and feature rich tools for performance monitoring. Some of these tools are open sourced.

The majority of such tools use Java bytecode instrumentation technology as a means to probe the application's state and gather data. To give credit to the technology an overview of related work is provided next.

2.1 Related work

There is quite an extensive amount of research done on the topic of bytecode instrumentation (BCI). Most of the academic papers related to BCI focus on the performance and recording full calling context trees (CCT) which are basically a traces of program executions. A CCT provides a hierarchical structure of executable units such as methods, controllers etc. An example of a possible visual representation of CCT on Figure 1**.**

FERRARI (Framework for Efficient Rewriting and Reification by Advanced Runtime Instrumentation) is a tool which provides instrumentation coverage of all classes loaded by JVM including JRE core classes and native methods. To instrument native methods and some core JRE classes the tool uses static instrumentation. As these methods and classes are loaded into a class pool only once then FERRARI tool runs instrumentation process during bootstrapping of JRE before actual application execution. In this work, there are several descriptions how FERRARI's approach overcomes some technical problems which are related

to static instrumentation and better calling context tree (CCT) gathering. The work was published in 2007 [7] and achieved better instrumentation performance than JVMTI.

InsECT (Instrumentation, Execution, and Coverage Tool) [8] tool which predates FERRARI. As the work was published in 2004 there was not Java support for instrumentation process. The tool relied on custom classloader to do BCI before and after bootstrapping of JRE classes. InsECT also features static and dynamic instrumentation although it does not support instrumenting all classes. InsECT depends on another tool featured in this work PTrace which analyses information.

MAJOR [9] is a tool developed in Switzerland at the University of Lugano. MAJOR is an aspect weaver that complements most popular aspect viewing framework AspectJ. It is also a tool which heavily relies also on previously mentioned FERRARI a bytecode instrumentation framework which focuses full method coverage. Most relevant to current thesis topic were the description of process of generation CCT process generation and latency measuring with AOP approach.

Another article [10] introduces several tools which are builds upon MAJOR. As MAJOR introduces additional functionality in the area of AOP which by itself leverages instrumented application, several projects emerged. Firstly **DJProf** which is a tool to generate complete CCTs, secondly a tool which detects memory leaks based on registering all instance allocations and heap memory analysis and the last tool was **ReCrash** which generates a test based event such as exception thrown in code

Takipi [11] is a tool similar to ReCrash on the market is. It concentrates functionality around showing code with evaluated state before an exception is thrown.

2.2 Application performance monitoring tool market overview

The review provides an additional context for the framework introduced in the current thesis. The framework uses similar technology to probe the application to gather data.

What is application performance monitoring (APM)? APM provides features which automate and reduce effort to monitor and track the performance of the application and usability metrics. Such tools feature statistical representation of performance metrics such as response times, SQL execution times, throughput, JVM stats etc. In addition to statistical metrics APM tools often do log management and analysis; code drill-downs about thrown exceptions by Java, reporting downtime etc.

This subtopic provides a brief overview of most known APM tools available with the comparison of the features and products in compact overview in Table 1. This review focuses only on tools for Java ecosystem and the ones which leverage most Java BCI. Most of the listed tools target web application market.

If one comparing the reviewed tools to the framework introduced in the current thesis, there is a difference in the approach. The reviewed APM tools focus on detecting performance and usability issues on production environment while the current framework focuses on keeping integrating performance related test into the development loop by integration into continuous integration loop to reduce or eliminate serious issues at the earliest possible stage.

The features and functionality of the appropriate tools for comparison is presented in Table 1

- Requires source code modification – It is a strong deterministic factor for required effort to integrate an APM tool into application.
- Calling context trees and application layer diagnostics – Very important for troubleshooting and detecting root cause of the problem. This functionality is required for the framework presented in this thesis.
- Cross domain transaction tracing – Critical functionality if components of the software system are distributed.
- Application topology mapping – Provides a broad overview of the whole system.
- Notifications – Reporting in case of an incident such as downtime of the whole system or a component of the system.

- API – Provides access to data such as service health notification and different high level metrics. An important factor for integration with project management software or company's inside management tool.

Table 1 AOP tool functionality overview

	Requires source code modifications	Calling context tree recording & application layer diagnostics	Cross domain transaction tracing	Application topology mapping	Notification	Notification and metric API
AppDynamics	No	Yes	Yes	Yes	Yes	Yes
NewRelic	No	Yes	Yes	no	Yes	Yes
DynaTrace	No	Yes	Yes	Yes	Yes	Yes
Plumbr	No	Yes	Yes	No	Yes	Yes
X-Rebel	No	Yes	Yes	No	No	No
Pinpoint	No	Yes	Yes	Yes	Yes	No
Stagemonitor	Yes	Yes	No	No	No	No
Java melody	Yes	No	No	No	Yes	Yes

Table 2 APM product overview presents the APM tools from the maturity and business point of view. In addition, the table describes an effort which has to be made to create mature and usable APM tool. During this thesis, an effort[1] for creating such tool was made. The tool

[1] Even for an open source project it took about two years for to be in a state to be usable with different technologies. For a startup, it took millions of funding and many on average 2 more years of development compared to open source alternatives.

worked only with a smaller example application, it could not handle larger application which also included other bytecode manipulation tools and used AOP. After recognizing the required effort and realizing the limitation such custom made tool with compliance issues could imply, another approach was chosen. In addition amount of funding reflects the market need of application performance monitoring.

Table 2 APM product overview

	Business model	**Founded year & total funding**	**Price (April 2016)**
AppDynamics	SAAS & on premise	2008	Custom pricing.
NewRelic	SAAS only	2008	1788$ annually
DynaTrace	SAAS & on premise	1993 21,9M	Contract based from 1200$ to 10000$ annually [12]
Plumbr	SAAS & on premise	2011 1.7M [13]	1008$ per JVM annually
X-Rebel	SAAS	2014	365$ per Developer annually
Java melody	Open source	2009 Many active contributors	Apache 2.0 License
Stagemonitor	Open source	2013 Mainly one company supported active contributor	Apache 2.0 License
Pinpoint	Open source	2013 Several company supported active contributors	Apache 2.0 License

2.2.1 AppDynamics

AppDynamics provides an APM tools for wide range of technologies. Current review focus only on Java technologies.

Java application performance monitoring by AppDynamics provides monitoring for full software stack from JavaScript by injecting alien probing code on the front end to the very depths of DB layer by analyzing database logs files. AppDynamics features a topology graph to show end to end transactions. Tool features as well a possibility of error stack trace drill-downs to ease detection of the root cause of the problem.

Tool promotes ease of use and faster integration. [14]

2.2.2 NewRelic

NewRelic is an application performance monitoring tools for wide range of technologies. Current overview will focus only on Java technologies.

Similar to AppDynamics, the tool collects metrics from front-end and back-end components. Instead of topology graph, it uses application map to determine the status of services. Service statuses are segmented into three separate categories (satisfied, tolerating, frustrated) based on response time and customizable threshold values. Otherwise, it is very similar to AppDynamics. [15]

2.2.3 Dynatrace

Dynatrace is an application performance monitoring tool for a several technologies. This overview will focus only on Java.

The tool features similar functionality to tools listed above and especially focuses on infrastructure perspective mainly from the application's usability point of view. Features such as metrics from remote servers (micro service architecture) with a rich network, application, database and JVM metrics. All the information is displayed on (application topology map) node map where each node is spate system component. As Plumbr, AppDynamics and NewRelic this tool also features detecting performance and service availability issues in real time on a production environment. The tool also provides notification of such issues and information to detect the root cause of the problem. [16]

2.2.4 Plumbr

Plumbr is an APM tool developed in Estonia. It was initially introduced as a memory leak detector. BCI is used to track allocation of objects and when they're freed with by the Java garbage collector. Plumbr also has also a collector service where the memory leak data is analyzed. The collector analysis service learns over time to recognize common memory leak problems and provides solutions from constantly improving database. Plumbr's memory leak detection feature focuses on avoiding false negative report of memory leaks.

In addition to memory leak detection, the tool offers very similar functionality to tools mentioned above. Cross-domain transaction tracing, metrics, notifications etc. Plumbr tool lacks software topology mapping view. [17]

2.2.5 ZeroTrunaround's X-Rebel

X-Rebel is ZeroTurnaround's third outstanding product in series. The tool is developer oriented and is intended to be used only during the development process as it provides information about metrics per single user session. The tool has a user interface named ninja UI where call context transactions are presented. The user interface is added to the presentation layer as a button with some metrics and warning signals based on customizable thresholds, which similar to a single page web application as a popup. In the case of an application without presentation layer, (for example REST services) ninja UI is presented as a separate context path.

Latest release 3.0 focuses of cross domain calling context tree tracing in micro service architecture. That means that full trace of relevant information such as used services on application layer and evaluated formatted SQL queries will be presented even if program execution point moves through another micro service on remote server with attached X-Rebel javaagent. [18]

2.2.6 Java melody

Java melody is an open source tool which presents statistical information for Java and Java EE applications. The tool could be used in production as well as in development environments. The

tool provides a user interface with rich graphs and charts and also offers an API for accessing the statistical metric in JSON or XML format. [19]

The tool provides statistical information about SQL queries, HTTP request response time statistics and also metrics about JVM such as memory and garbage collector metrics.

A shortcoming of this monitoring tool is the lack of calling context trees. The tool provides too little information about the application layer. This is the main reason why this tool could not be used as a probing service for the framework.

For integration JavaMelody requires additional dependencies in the source code of the application such as declared API dependencies (for example in Maven's pom.xml file) and some lines of code for configuration.

JavaMelody is used as an APM tool for AIO system which is described in further Section 5.2

2.2.7 Stagemonitor

Stagemonitor is an open source APM tool developed in Germany which started as a master's thesis topic. Stagemonitor is based on BCI and is heavily integrated with Elasticsearch for data retention and uses Graphite or Kibana for data visualization. This tool is something in between of XRebel and JavaMelody. Like ZeroTurnaround's X-Rebel it has additional context path for single page interface to display in bowser widget which is used to present latest HTTP request and JVM metrics. On the other hand Stagemonitor features visualization of all request data with rich graphs and charts in Kibana [20] [21]

2.2.8 Pinpoint

Pinpoint is an APM tool developed in South-Korea and supported by a company called Naval. Pinpoint features functionality which is very similar to expensive tools such as AppDynamics, NewRelic and DynaTrace. Taking into account that the tool is open source and has been developed for 4 years with release number 1.6, in my personal opinion, it is a quite considerable choice for an APM tool for production monitoring. [22]

The reason why this tool was not chosen for framework the current profiling tool was because it did not have a persistent metric storage. Although data containing metrics could be obtained from the tool via JMX.

2.3 Tool choice for current framework

Stagemonitor was chosen as the application probing tool for the current framework.
Firstly the choice was narrowed down to Java Melody, Stagemonitor and Pinpoint. All these tools are open sourced. Java Melody was eliminated from the choice because it did not collect as much information from the application layer. At this point, only Stagemonitor and Pinpoint remained. The choice was narrowed down to Stagemonitor because of two reasons.
Firstly, the ease of integration. Stagemonitor was much easier to integrate because the data was persistent and easily accessible with queries. In the case of Pinpoint a separate module had to be created to make data easily accessible.
Second most important criterion was the complexity of the system. Stagemonitor was a smaller monolithic application with attachable modules. On the other hand, Pinpoint required several dependencies to be running.
So a simplest open source tool was chosen because only collecting calling context trees and other performance related data was required.

3. Java bytecode instrumentation

A computer program instrumentation refers to the ability to monitor the execution of the program. It is often used for diagnostic purposes such as logging and measurement data recording at run time. As the instrumentation allows data gathering at run time, it gives better opportunities information and meta-information collection compared to program code static analysis. [23]

Definition of word *instrument*:
"To equip with instruments especially for measuring and recording data" [24]

In Java, instrumentation as a technique is used in the context of Java bytecode manipulation. Because Java is a higher level programming language there is a very specific abstraction in the process which needs to be explained. The normal compilation procedure of Java takes the source code in Java (stored as a .java files) to compiled .class files which contains bytecode. Java program execution uses the bytecode which is interpreted (and compiled in the just-in-time manner) by the JVM to executable code for specific hardware. During the execution of a Java program, the Java classes are dynamically loaded by class loaders, so the process is dynamic and relies on program runtime execution coverage.

Java bytecode instrumentation is a process of bytecode manipulation during the phase of program execution – at the time Java class loading. Manipulation consist of reading bytecode and conditionally changing the bytecode to result in the desired change.

Essentially Java bytecode instrumentation (further JBI) grants the ability to add functionality without modifying application's code. The functionality is available since Java 5. [25]

3.1.1 Use cases in general

Transforming classes without modifying source code

- **Code execution profiling** – Most APM tools by companies such as Plumbr, AppDynamic, NewRelic offer products for Java often as a service, which is heavily relying on BCI for acquiring metrics of the state of the application. These services are often attached to client's Java application on the in the production environment.

  Profiling the state of the application is a technology which all the reviews application performance monitoring tools in the previous section 2.2.

- **Code optimization –** There are several Java bytecode optimization frameworks such as e.g. SOOT [26].

Program analysis

- **Static code analysis, bug detection** – Such tools use static code analysis on bytecode level. Most known is FindBugs which uses ASM and BCEL [27]

- **Measurement of code complexity and coverage** – Java program execution takes place at bytecode level so the BCI techniques are used for analysis. Bytecode instrumentation is used for recording program execution coverage details such as method coverage, branch coverage, LOC coverage, variable use etc. Bytecode instrumentation is required Java program runtime on bytecode level. [28]

- **Matching classes with specific annotations** – This technique is gaining popularity since Java 1.5 release which introduced annotations. Annotations are used heavily in AOP and in most widespread Java frameworks as e.g. Spring.

Generation classes

- **Lazy loading data from database using proxy pattern** – Best showcase for the use case of such technique is an ORM like Hibernate which uses lazy loading of entities. When the attribute is called and the entity has not been initialized, then a spate JDBC call is made to fetch data from the database and the bytecode of already initialized class is loaded by a bytecode provider. [29]

Below is an example from a model class of widely used ORM framework Hibernate. Annotation presented below will be picked up by Hibernate that the entity object will be initialized with using BCI. [30]

```
@OneToOne(fetch=LAZY)
private Entity entity;
```

Security

- **Obfusticaton of code** – Compiled Java classes are by default easily reversed. Decompiled Java classes are usually human readable and serve high risk of software as a property loss. The issue is especially relevant in the case of client-side application. BCI offers the solution to additional complexity for information loss during a decompiling process. [31]

- **Access restriction to APIs -** As an additional measure to Java OO open closed principle for access restriction which is easily overcome by Java core functionality – reflection. [7]

3.1.2 Tracing calling context trees as an use case

Calling context trees (CCT) provides means to debug and analyze software performance issues in terms of interconnectivity and causality (root issue detection). For CCT tracking, BCI provides a well-suited functionality for gathering additional information about program execution without modification of original source code. It is a very common use case for BCI. The downside of CCT tracing is high performance overhead. Adding additional instructions before and after every method call and sending output data to external collector service after each method call increase the overhead especially in a case of smaller methods such as utility methods.

To solve the overhead problem Java's a `ThreadLocal` [32] class could be used. By storing instead of sending each calling context related data gathered by instrumentation into the stack and making the data independent upon each other by grouping the scope of variables

containing instrumentation data by Java unique thread identifier. Reconstruction of CCT-s from the `ThreadLocal` storage has the following aspects:

- Mapping calling context initiation event such as HTTP servlet request to data from the `ThreadLocal` storage. By assigning a unique identifier to a thread at an endpoint of the call, data collected by instrumentation could be grouped by the unique thread identifier held in the `ThreadLocal` temporal storage.

- Parsing calling context data such as argument types with the values and timings from the `ThreadLocal` storage.

- CCT has to be compiled into a hierarchical structure for better representation based on needs of a consumer. As the individual method call data could be stored in the `ThreadLocal` the order of the actual calling context unit (for example and method call) execution data could be mixed. The ordering of CCT traces could be done by a sequence number or a timestamp.

- Managing load by using the `ThreadLocal` storage as a queue. The storage could be used as a queue with consumer such a collector service and an instrumentation service as producer.

In a case of micro service architecture, transactions could be traced over the network by adding a wrapper to the endpoint services. [33]

3.2 Standard Java Instrumentation support

Improved instrumentation functionality support was added to Java language in version 1.5. [25] The update introduced features for supporting instrumentation APIs. The older version of Java Bytecode instrumentation 1.4 and below JVMTI and Retroweaver tool [34] could be used. [23] [35]

Below is a short description about running compiled Java program without attached Java agent.

```
java ee/ttu/example/MyProgram
```

The `MyProgram.class` goes into a system classloader. `MyProgram.class` gets loaded. Program execution starts from main method with following which is looked up by the classloader: `public static void main(String args[]).`

Following example describes execution of a Java program with attached javaagent

`java -javaagent:/path/to/agent.jar ee/ttu/example/MyProgram`

1. From jar-file manifest the permain following method public static void `premain(String args[], Instrumentation inst)` from `Agent.class` is called.
2. Inside a `premain` body JVM registers a `ClassFileTransformer` interface implementation instance, let's use MyTransfomer.class which implements following method from the interface:
   `byte[] transform(…, byte[] bytecodeToBeLoadedClass)` which processes the byte code during class loading phase. Every class loaded by JVM for particular Java program goes through this method.

   Method signature of `ClassFileTransformer`

```
public byte[] transform(
  ClassLoader loader,
  String className,
  Class<?> classBeingRedefined,
  ProtectionDomain protectionDomain,
  byte[] classfileBuffer
)
throws IllegalClassFormatException {
  …
  return modifiedClassfileBuffer;
}
```

   loader – The classloader which loaded the class.
   className - The name of the class. For example "java/util/String"
   classBeingRedefined - This argument is evaluated if the class is intended to be retransformed (if the class is already loaded) or redefined (changes actual definition of already loaded the class). Value null if the class is just loaded.
   protectionDomain - the protection domain of the class being defined or redefined
   classfileBuffer – The actual class bytecode data in byte array format.

3. After transformation the bytecode of `MyProgram.class` is changed and program execution starts from calling `public static void main(String args[]).`

The behavior will be different compared to the unmodified class in case of BCI yielded any results.

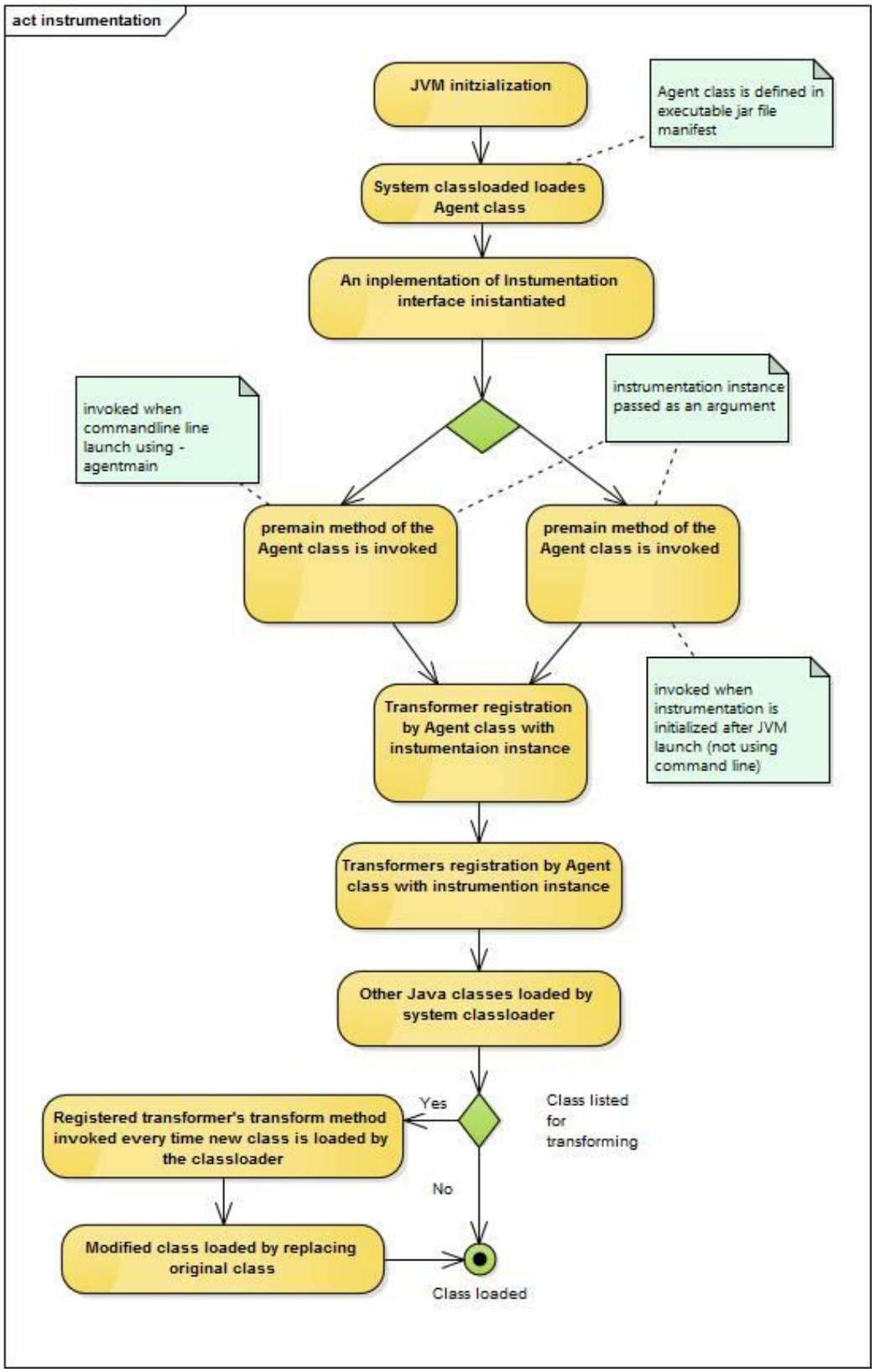

Figure 2 Java standard instrumentation

In Figure 2 the class loading is visualized including two flows to initialize a transformer in premain method. The first approach is as described in subsection 3.3.2 by adding javaagent argument to the VM arguments and the second flow in plain instrumentation initialization by

Java programmatically. In the diagram, there is a condition which filters classes from bytecode transformation.

3.3 ASM

3.3.1 Abstract

ASM is a tool for dynamic manipulation and creation of Java class files. ASM is known for its performance and rich low-level functionality. The tool uses an approach of visitor design patterns to filter classes which gives an advantage in performance. ASM is one of low level tools suitable for BCI which is considered as a bytecode manipulation standard. One of the reasons is support actual byte code opcodes editing which is not as human readable as the Java code. Also, the library is very lightweight. ASM 5.0.2 contains only 25 classes and the dependency is only 52KB in size. [36]

3.3.2 Description of setup

The event based approach for BCI was described in the previous chapter 3.2. Inside the transform method a `ClassReader` instance has to be initiated, then the reference of the instance is passed as an argument to an initialization of `ClassWriter` which is responsible for creating and optimizing class bytecode. All the not human-readable instructions for bytecode manipulation are in the `ClassVisitor` implementation which actually deals with manipulating the bytecode. [37] [36]

```
ClassReader cr = new ClassReader(classfileBuffer);
ClassWriter cw = new ClassWriter(cr, ClassWriter.COMPUTE_FRAMES);
ClassVisitor cv = new AnnotatedClassVisitor(cw, className);
cr.accept(cv, 0);
byte[] byteArray = cw.toByteArray();
```

3.4 Javaassist

Javaassist is a bytecode manipulation framework. It provides higher level in addition to lower level API for bytecode manipulation. Compared to ASM the Javaassist bytecode manipulation

instructions are rather human readable as the code which is injected to bytecode is valid Java code represented as a String. For example following

```
StringBuilder sb = new StringBuilder();
sb.append("System.out.println(\"Hello javaassit\")" );
method.insertBefore(before);
```

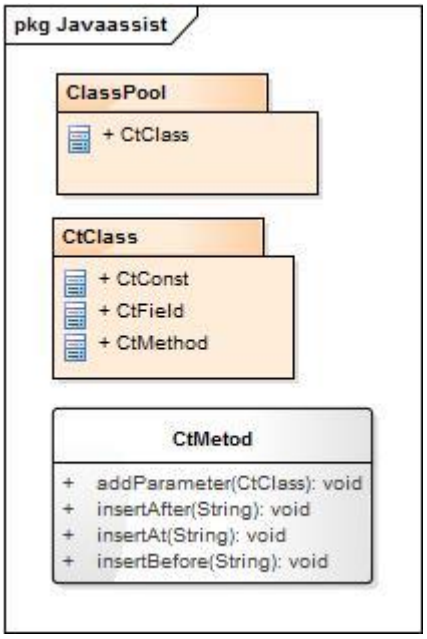

Figure 3 Javaassist

On Figure 3 We can see that Javaassist uses `ClassPool` to obtain a class. A class is represented as `CtClass` type which contains attributes. For example, on attributes is a method representation as a `CtMethod` type. In `CtMethod` class, we can see some method representing the behavior of the class.

3.5 Summary

The current chapter provided a brief overview of most used technologies for bytecode manipulation. It is important because all the application performance monitoring tools described in subchapter 2.2 relied heavily on this theology. As the tool for probing the application in the subchapter was chosen to be open sourced, it is important to understand underlying technology. In case of chosen tool – Stagemonitor, Javaassist was used. Compared to ASM Javaassist provides a higher level of abstraction and the instruction for manipulation bytecode are human-readable, which makes it easier to work with.

4. Performance evaluation framework

Application performance management has been around for a while. Previous chapters introduce relevant academic work in Section 2.1, APM tools market overview in Section 2.2 and most important the latest technological advances in the context of BCI in Chapter 3 which made possible gathering highly valuable data for performance troubleshooting.

In Section 2.2 descriptions of tools which deliver high value in terms of production environment monitoring or troubleshooting errors during development are provided.

What is missing from the bigger picture of performance monitoring and evaluation tools? As is widely known fact that the cost of errors grows exponentially as the software matures and gets adapted by users. The situation is equally severe in the case of performance related non-functional issues which prevent the scalability and the usability of the application.

4.1 Overview of the components

In current thesis a framework for empirical evaluation of Java application performance is introduced.

The framework approach for performance evaluation concentrates on elimination performance issues at the earliest stage, i.e. during development. The framework utilizes specialized load testing for simulating application's usage and collects performance data for each service. The data is later analyzed and then the results are summarized and presented to the user.

To detect performance problems related to scalability during the load generation phase, it is specialized to achieve the following outcomes. Firstly, to simulate the accumulation of data for a specific user periodically over multiple times and secondly simulate many new users without periodical usage data. The first method of load generation targets issues which are related to *usability* issues and second approach targets performance issues which often limit

scalability such as e.g. indexing. Both types of load generation are implemented for AIO application in further Section 5.2.

During the load generation, the application's source code is instrumented and a profiling agent is collecting and storing performance related metrics which will be later used by an analyzer service.

For load generation data is collected and stored after every modification of the source code of the application.

Finally a data analyzer processes the data stored for every modification and compares the processed results to the previously processed results and presents the output summary.

4.2 Deployment process

The following order of processes has to be integrated into a continuous integration tool such as Jenkins.

1. Deployment of the application with attached profiling agent
    a. Resetting the state of the application to initial state.
2. Launching the load generator (post build task [38])
3. Launching the data analyzer (post build task)
4. Making a source code change
5. Repeating this process from step 1.

The process is detailed below in section 4.5.

4.3 Process comparison with load testing process

To provide additional context for the framework this section will summarize the main differences of the current process and previously known load testing.

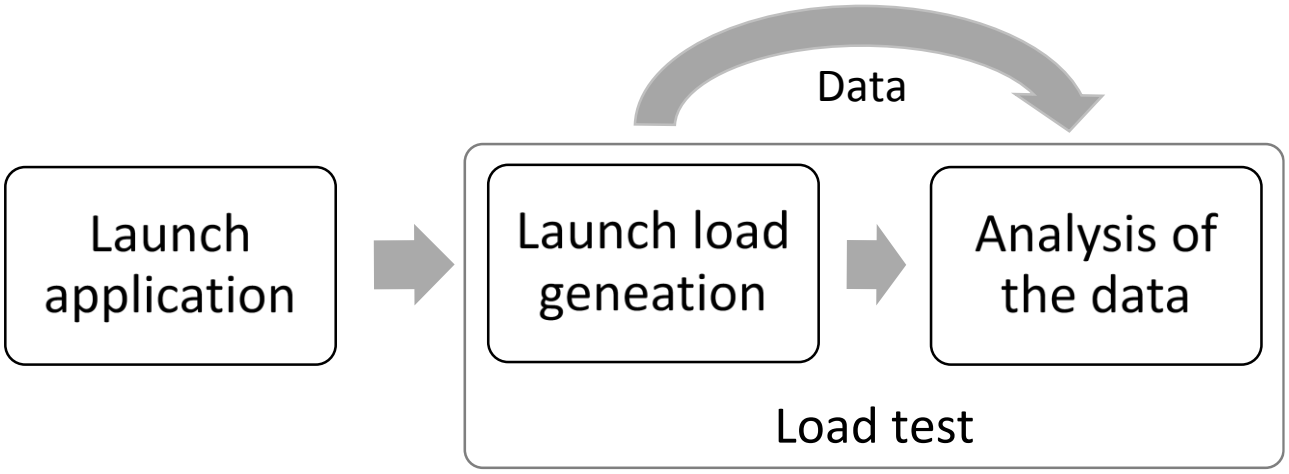

Figure 4 Load testing process

In Figure 4 there is a segment of a simplified process of load testing. Firstly an application under testing is launched, then the load generator is launched and during or after the load generation process the data is analyzed. Most commonly data consisting of response times and response statuses is collected and presented in a statistical manner.

Most import detail to notice in this process is that data to be analyzed is collected from the load generator. This approach settles has certain limitations. The data could be collect only from controller level. Only a small chunk of data about the state of the application is exposed via controller layer that could be used. Data such as response code or content type.

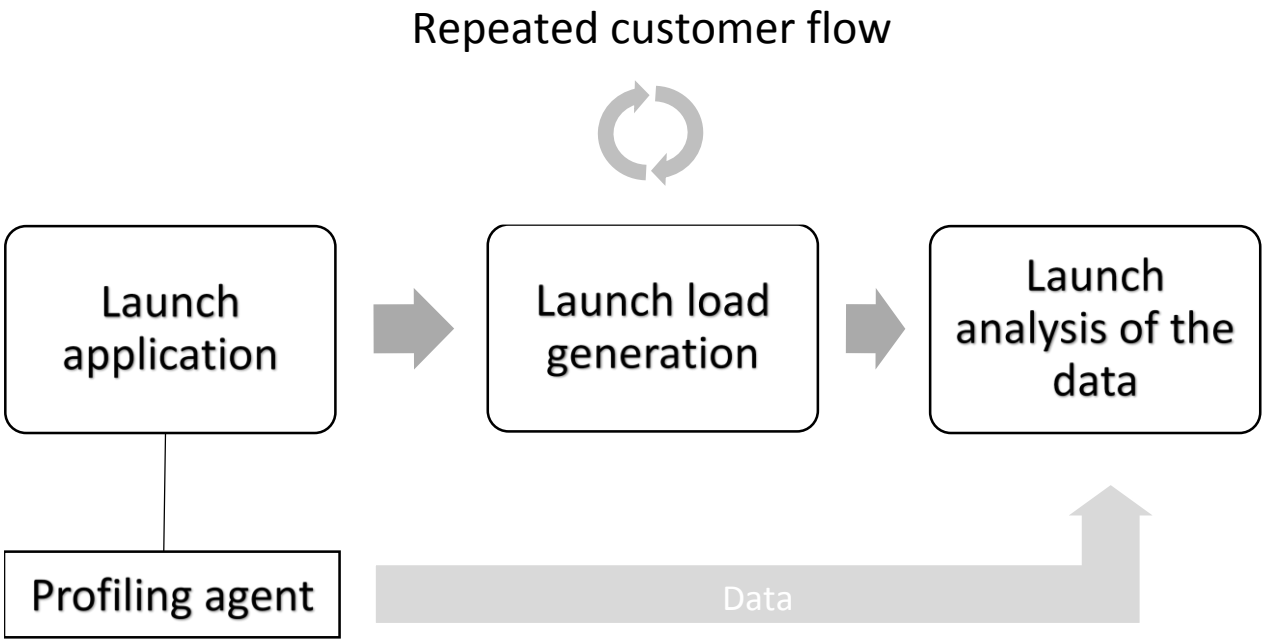

Figure 5 Framework's process

On Figure 5 is a simplified process of the framework. An application under evaluation is launched with attached profiling agent. After that load generator is launched which repeats repeatable segment from customer flow to progressively stimulate accumulation of state of the application. After the load generation process has finished the data analyzer is launched.

The most important difference compared to conventional load testing is that the data is collected from inside of the application which is under evaluation. It distinguishes process by a principle. Data collected from the application layer contains much more information which could be used for analysis of possible performance problems. For example in addition to response time and status codes the calling content trees with a list of all executed application layer timed services, database calls e.g. could be used.

4.4 Architecture

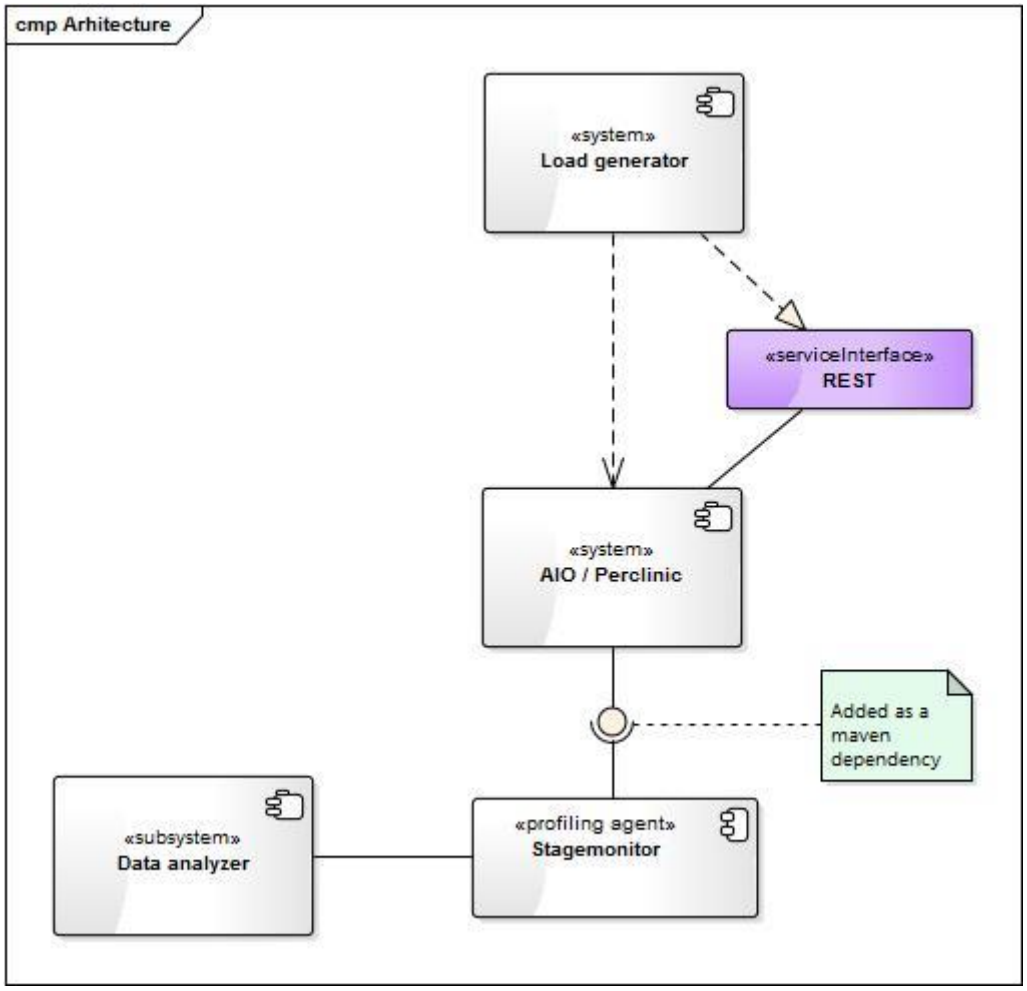

Figure 6 Framework architecture

On Figure 6 are several elements. Firstly an application under analysis. Then the load generator which is dependent on the source code of the application[2]. The application exposes resources via REST services which are consumed by the load generator. To the application a profiling agent - Stagemonitor is attached which instruments Java classes and collects and stores performance related data. For last there is a data analyzer which processes stored performance related data and outputs results.

[2] Load generation dependency upon source code of the application is to reduce the effort for the creation of application specific load test.

4.4.1 Load generator

For a load generation various tools could be used. For example JMeter or The Grinder [39] [6]. A custom load generation API is provided in this thesis work.

This thesis presents a custom tool because the case-studied application which is described in Section 5.2, required a lot of application-specific models for creating load generation scripts. A custom API was developed to reduce the effort of creation application specific commands to initiate required HTTP request to imply load on the application. In the specific case of AIO, the conventional load generation methods such as the use of the JMeter would require a lot of application specific models which had to be created from scratch.

The custom load generation API is based on well-known Java RESTful application client Jersey. [40] The API features easily specifiable request builder with already adjusted compliance with the analyzer. Most importantly API features easy integration with a specific application.

4.4.1.1 Load generation methodology

Before load generation process a state of the application is reset to the initial with database scripts. Load generation process is executed in a standalone machine to reduce side effects from varying environment factors.

To find performance issues a method of accumulation state specific data is used. For example, customer flow is repeated multiple times.

4.4.1.2 Connectivity with data analyser

Data analyser depends on data generated by the load generator which is stored in HTTP request headers.

The method is good because the majority of load generators available support customization of headers. In addition, it implies relatively low HTTP traffic overhead.

The following table represents the meaning of headers which are added to the HTTP request initiated by the load generator.

Table 3 Load generator specific HTTP headers

Requests identified by analyzer by following HTTP headers: HTTP header name	Format	Description
request-name	000_000000_0000	Combined identifiers to following collected metrics: session-id, request-id and period-number
request-id	000000	Every distinct request get a unique value. Specified in the request
session-id	000	Session identifier which represents use case session. For example new customer go unique session-id.
period-number	0000	In the case of periodic load, generation types the value is incremented on each new period.
modification-id	0000	Unique identifier which specifies the data set of metrics for specific application's build

4.4.2 Profiling agent – Stagemonitor

Stagemonitor is an open source APM tool which was chosen to be used in the current framework in Chapter 2. The tool suits in the context of the framework presented in thesis for its modularity and data gathered from profiled application.

Stagemonitor loads the javaagent dynamically at run time of the application. During class loading, the tool instruments loaded classes selectively with a chain of responsibility patterns. Data from instrumented classes is stored in Elasticsearch in JSON format. An example of the data is available in Appendix 1. For data visualization, Stagemonitor features two approaches. Firstly there is an in-browser interface for showing last HTTP request information and JVM metrics that could be used for developer side manual testing for a single user session. Secondly, there is the possibility to visualize all the session's data in Kibana [20]. Stagemonitor also includes around two dozen of preconfigured graphs and several graph dashboards to provide data visualization as an out of the box feature in Kibana.

4.4.2.1 Architecture overview

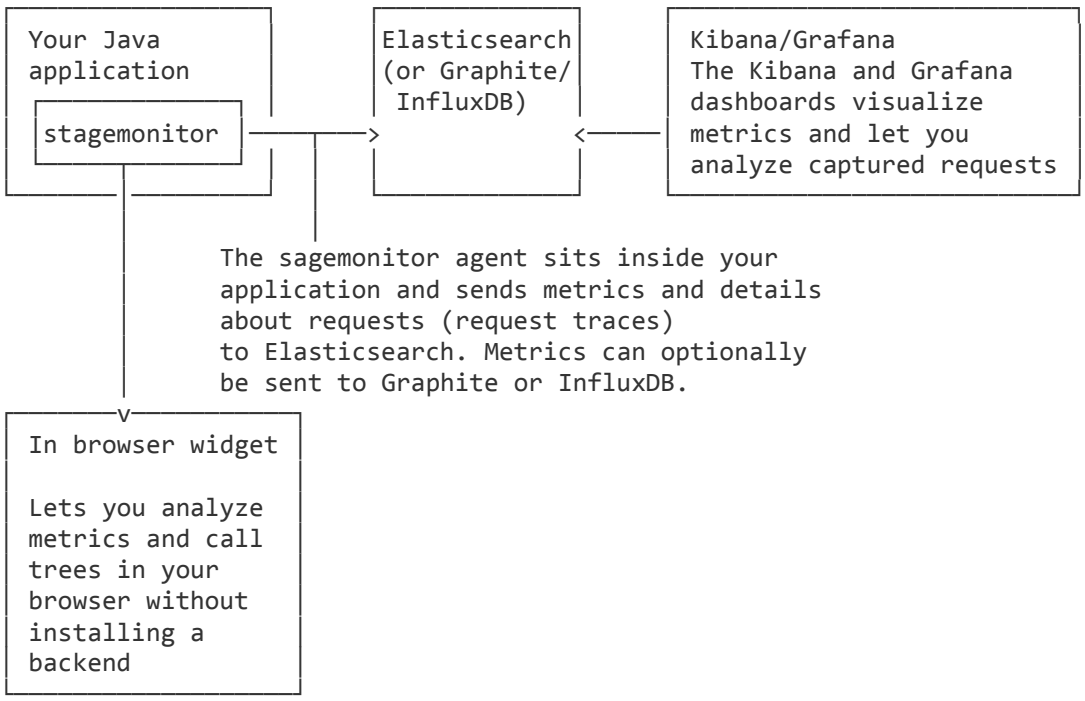

Figure 7 Stagemonitor architecture

On Figure 7 is association diagram of Stagemonitor components. The diagram is from Stagemonitor's wiki on Github [21].The centric component is Elasticsearch [41] where data from the profiling agent is stored. In current thesis, only minimalistic configuration is used. The only Stagemonitor the profiling agent and the Elasticsearch component. More about configuration in Stagemonitor repository documentation [21].

4.4.2.2 Setup

For integrating Stagemonitor into an application two dependencies have to be added to the source code of the application.

Stagemonitor maven dependency on Figure 8 which must be placed in application Maven's configuration file pom.xml and a stagemonitor.properties file which have to be placed in source code resource folder.

To enable data retention and data visualisation. Elasticsearch and Kibana have to be running before launching the application with Stagemonitor. Default ports and host is used.

Following properties have to be specified in the stagemonitor.properties file:

- stagemonitor.elasticsearch.url= http://localhost:9200
- stagemonitor.applicationName=aio
- stagemonitor.instanceName=loanengine

More about setup available at Stagemonitor repository wiki page. [21]

```
<dependencies>
    <dependency>
        <groupId>org.stagemonitor</groupId>
        <artifactId>stagemonitor-web</artifactId>
        <version>0.23.0</version>
    </dependency>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>jstl</artifactId>
        <version>1.2</version>
    </dependency>
    ...
</dependencies>
```

Figure 8 Stagemonitor maven dependency

4.4.3 Data analyzer

The data analyzer processes the request data persisted to ElasticSearch [41] by the profiler agent – Stagemonitor. The profiler agent persists request data which is induced by the load generator.

The request data is grouped by modification identifiers and every modification identifier got a group of request identifiers. For example:

- modificationId - 0000
    - requestId – 001000
    - requestId – 002000
- modificationId – 0001

    - requestId – 001000
    - requestId – 002000
- …

For every unique request identifier and modification identifier a pair of ElasticSearch queries is made which retrieve all request data for specific source code modification and specific request type (For example specific REST service).

Due to specific load generation, the customer flow is repeated during load generation. So basically we got a set of request data for similar the same services but with a different state of the application. For example database state differs. Request data size is equal to the periods count customer flow was repeated.

Analyzer uses the full request data (an example is given in Appendix 1). Only the attribute used by the data analyzer is represented in Table 4.

Table 4 Attributes used from Stagemonitor request data

Attribute name	Example of value	Description
executionTimeDbt	292	Time spend in database layer (executing SQL statements) [ms]
executionTime	564	Total request time [ms]
executionCountDb	23	Number of executed SQL queries during request
executionTimeCpu	342	Time spend in application layer [ms]
callStack	Figure 1	Calling context tree.
callStack.size	723432	Length of CCT representation as a Java String type
callStack.depth	172	Number of rows in calling context tree

For every attribute, the metric value summarized shown in Table 5 is calculated. Design-wise a pattern of responsibility was used. For every processor, there is a separate implementation code wise.

Table 5 Metrics calculated for each request data attribute

Metric	Description
Average	Arithmetical average of attribute value
Median	Media of attribute value
Min	Lowest occurred attribute value
Max	Highest occurred attribute value
Range	Difference between attribute's maximum and minimum
Diff average	Rate of change - Average of difference between current and previous attribute value
Square root average	Average of square the roof of difference between current and previous value the absolute value
Standard deviation	Standard deviation

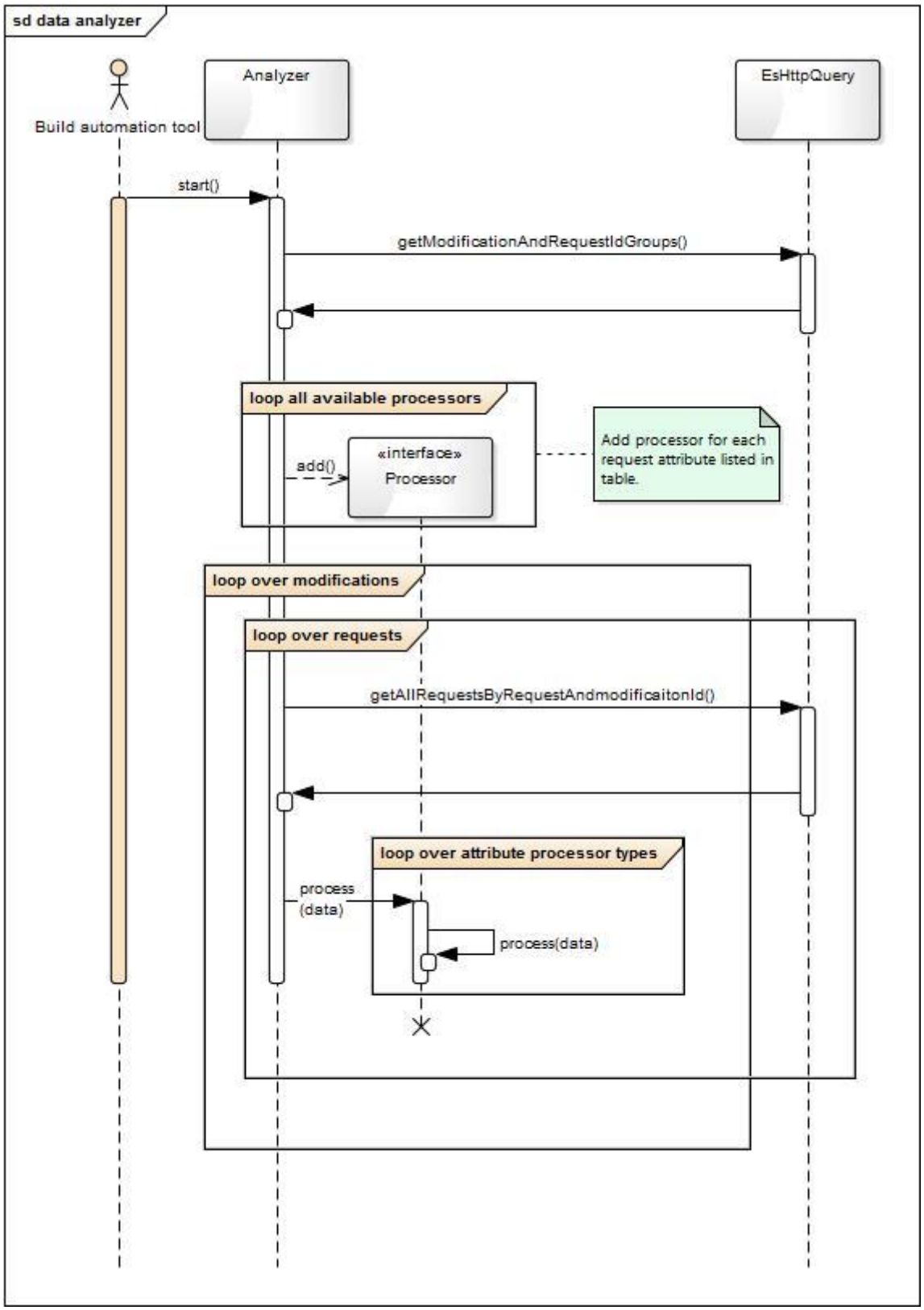

Figure 9 Analyzer component iteration diagram

Figure 9 presents an iteration between different components. Description in topic 4.4.3

4.5 Setup

All configuration and source code required for using the framework is available in an open repository on Bitbucket. [1]

4.5.1 Launch an example application – **Petclinic**

The following list on command fetches source code from the remote repository and launces application Petclinic in already initial state because of an in-memory database.

```
mkdir petclinic
cd petclinic
git clone https://@bitbucket.org/viktor_reinok/petclinic.git
mvn clean tomcat7:run
```

4.5.2 Launch the load generator

The following list of commands fetches load generator source code from the remote repository, build the executable .jar file and launches the executable – load generator for Petclinic application with modification identifier 0000.

```
cd..
mkdir thesis
cd thesis
git clone https://bitbucket.org/viktor_reinok_thesis_team/thesis.git

mvn -pl load-generator -am package assembly:single -DskipTests -P load-generator-build-
profile
cd load-generator/target
java -jar load-generator-jar-with-dependencies.jar 0000
```

4.5.3 Launch the data analyzer

The following list of command build data analyzer .jar executable fail and launches the executable data analyzer.

```
cd../..
mvn -pl analyzer -am package assembly:single -DskipTests -P data-analyzer-build-profile
cd analyzer/target
java -jar analyzer-jar-with-dependencies.jar
```

4.5.4 Emulate the source code change and redeploy the **Petclinic** application

With the following list of command a commit is emulated which introduces an *N + 1 select* problem and restarts the PetClinic application with initial state.

```
stop existing deployed instance of Petclinic
cd../..
cd petclininc
git checkout abc4b24337c8fce97aa557620b8ad8d7e047a49a -f
mvn clean tomcat7:run
```

4.5.5 Launch the load generator again

The following command rebuilds and relaunches the load generator with different incremented modification identifier.

```
cd..
cd thesis
mvn -pl load-generator -am package assembly:single -DskipTests -P load-generator-build-
profile
cd load-generator/target
java -jar load-generator-jar-with-dependencies.jar 0001
```

4.5.6 Launch the data analyzer again

The following list of command rebuilds and relaunches the data analyzer which will output new information about introduced performance issue. Available in Figure 10.

```
cd../..
mvn -pl analyzer -am package assembly:single -DskipTests -P data-analyzer-build-profile
cd analyzer/target
java -jar analyzer-jar-with-dependencies.jar
```

5. Application of the framework

To proof the effectiveness of the performance evaluation framework developed in the current thesis, there are evaluations on two distinct case studies. Firstly a simple application which is publically available as a sample of potential problems in the Spring framework with Hibernate. Secondly a large enterprise scale application with hundreds of thousands of lines of code.

5.1 Simple sample application – Petclinic

First, we introduce is a simple sample application called Petclinic. The purpose of the original application is to demonstrate how to use the Spring framework. The framework also uses the most widely used ORM framework – Hibernate. Because of the use of ORM, there is a performance problem which limits the scalability of the application. The problem is widely known. The problem is known as *N+1 select query problem*. Basically, the N+1 select query problem is a situation where instead of an inner join query over two or more tables several simpler (single table with where id equals clause) queries are executed by an ORM framework. [29]

5.1.1 Results

5.1.1.1 Overview and setup description

The case study with a Spring framework sample application provides insight into the current framework for performance evaluation. The approach facilitates the validation of the effectiveness of each fix and detection of possible negative side effects the patches. In the case of the usual development process, the tool will highlight indicates new possible performance issues introduced in fresh code commits. The N+1 query problem is introduced during the development.

The application was launched with attached profiling agent – Stagemonitor and the state was reset. The load generator was set up to do 100 customer flow repetitions. After that an N+1 select query performance issue was introduced into the code by a commit. After the modification steps were repeated. A detailed description of the experiment is available in Section 4.5.

5.1.1.2 Result

As expected a performance issue was detected after code modification described in the previous subsection. After every customer flow, data was added to the application and during the read request each additional data entry was retrieved from the database with a separate SQL query, which means an N+1 select problem was introduced.

```
----------------------ModificationId 0000----------------------
Processing HTTP request metrics. Data set size   100. HTTP request: Request-id=001000 GET
/petclinic/owners
Processing HTTP request metrics. Data set size   100. HTTP request: Request-id=002000 POST
/petclinic/owners/new
Processing HTTP request metrics. Data set size   100. HTTP request: Request-id=003000 GET
/petclinic/owners/67
Processing HTTP request metrics. Data set size   100. HTTP request: Request-id=004000 GET
/petclinic/owners/72/pets/new
Issue diffAverage -> EXECUTION_TIME                value      0,18 Request-id=004000 GET
/petclinic/owners/72/pets/new
Processing HTTP request metrics. Data set size   100. HTTP request: Request-id=005000 POST
/petclinic/owners/67/pets/new
----------------------ModificationId 0001----------------------
Processing HTTP request metrics. Data set size   100. HTTP request: Request-id=001000 GET
/petclinic/owners
Metric CALLING_CONTEXT_TREE_SIZE      average            differed from last by  672,619%
Metric CALLING_CONTEXT_TREE_SIZE      median             differed from last by  338,838%
Metric CALLING_CONTEXT_TREE_SIZE      max                differed from last by 1205,945%
Metric CALLING_CONTEXT_TREE_SIZE      min                differed from last by  139,663%
Metric DB_QUERY_COUNT                 average            differed from last by  892,857%
Metric DB_QUERY_COUNT                 median             differed from last by  450,000%
Metric DB_QUERY_COUNT                 max                differed from last by 1600,000%
Metric DB_QUERY_COUNT                 min                differed from last by  185,714%
Metric EXECUTION_TIME                 diffAverage        differed from last by  237,500%
Metric DB_EXECUTION_TIME              average            differed from last by  192,388%
Metric CALLING_CONTEXT_TREE_DEPTH     average            differed from last by  367,647%
Issue diffAverage -> DB_QUERY_COUNT                value      0,93 Request-id=001000 GET
/petclinic/owners
Metric CALLING_CONTEXT_TREE_DEPTH     median             differed from last by  185,294%
Metric CALLING_CONTEXT_TREE_DEPTH     max                differed from last by  658,824%
Issue diffAverage -> CALLING_CONTEXT_TREE_SIZE      value    303,31 Request-id=001000 GET
/petclinic/owners
Issue diffAverage -> CALLING_CONTEXT_TREE_DEPTH     value      0,93 Request-id=001000 GET
/petclinic/owners
Processing HTTP request metrics. Data set size   100. HTTP request: Request-id=002000 POST
/petclinic/owners/new
Processing HTTP request metrics. Data set size   100. HTTP request: Request-id=003000 GET
/petclinic/owners/56
Metric EXECUTION_TIME                 diffAverage        differed from last by  200,000%
Metric CALLING_CONTEXT_TREE_SIZE      range              differed from last by 15600,000%
Metric CALLING_CONTEXT_TREE_SIZE      squareRootAverage  differed from last by  300,250%
Metric CALLING_CONTEXT_TREE_SIZE      standardDeviation  differed from last by 6920,527%
Processing HTTP request metrics. Data set size   100. HTTP request: Request-id=004000 GET
/petclinic/owners/52/pets/new
Metric DB_EXECUTION_TIME              range              differed from last by  150,000%
Metric DB_EXECUTION_TIME              max                differed from last by  150,000%
Metric EXECUTION_TIME                 diffAverage        differed from last by  138,889%
Processing HTTP request metrics. Data set size   100. HTTP request: Request-id=005000 POST
/petclinic/owners/52/pets/new
Metric EXECUTION_TIME                 diffAverage        differed from last by  133,333%
```

Figure 10 Data analyzer output - Petclinic

In Figure 10 is the output of the data analyzer is given. There are two datasets representing modification 0000 and 0001. Second modification dataset contains the comparison to previous modification results which start with prefix “Metric “. Underlined lines represent performance issues. Every additional customer flow step added data and because of that, an additional SQL query was made for every single data element added to the application. The framework detects it by computing the diffAverage parameter (detailed in Table 5) which indicates that every new customer flow step request resulted in an on average 0.93 SQL additional queries in the database layer. Thus the typical N+1 select problem was detected.

Hypothetically, if the performance problem was not introduced the second log with associated modification of 0001 would look exactly like the log with associated modification identifier of 0000.

5.2 Large case study: Information system - AIO

AIO stands for All-In-One. It is a server side application which supports a business critical microfinancing information system. The initial goal of the system was to unify common business logic and process to improve system maintainability. Unifying common business logic is especially critical for the system because it serves multiple markets which are all located in Europe. During marketing and advertising campaigns there are occasional spikes in load depending on the market. System has been operational for several years and customer base is constantly growing.

5.2.1 Abstract technical details

AIO has interfaces for a client and back office users. Server-side resources are exposed mainly via REST and SOAP web services. SOAP services and database replication are used for back office client and a single page user interface uses REST service. The system uses mainly Java based technologies. Software technology stack consists of Spring framework, Hibernate as a JPA provider and MySQL database. Infrastructure wise the system is divided mainly into two modules Client and Admin. Admin module is a SOAP web service endpoint and Client module which is a REST service endpoint. Both Admin and Client module have a common dependency which is included into deployable. Both modules are running in AWS cloud with a load

balancer. Client and Admin modules are in the separate load balanced physical servers. See Figure 11.

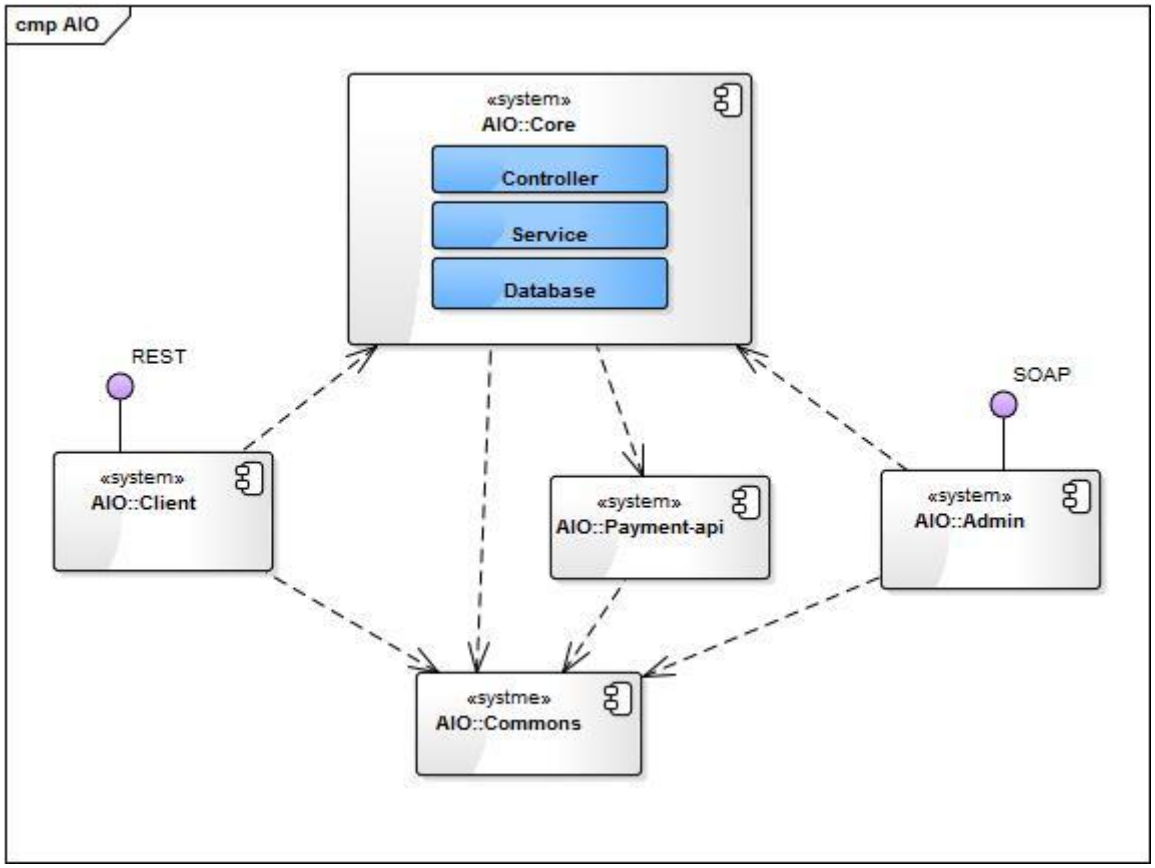

Figure 11 AIO modules

The AIO server side application has been developed by a team of 5-7 back-end developers for about 4 years. Project consists on mainly of 5 modules and there is around 160K lines of code.

5.2.2 Results

The framework introduced in this thesis detected several performance issues. Couple of the detected problems were known to exist before running the framework.

The framework detected many new performance problems which were unknown before. The problems were not as severe as the couple known performance problems or were located in optional segments of the customer flow which could explain the news. The nature of the problems implies that the performance issues will be experienced by customers with long usage

record. So the performance issues will affect customers who have been loyal to business for a longer period of time.

From a personal experience, it is obvious that the causes for detected problems are rooted deep in the core of the application and depend on many components which makes it hard to remove. By using the framework introduced in this thesis an empirical approach could be used to validate the fixes of issues and in addition, observe potential side effects caused by the fixes on performance.

Technical nature of the problem explains the issues with performance. Every new customer flow period generated by the load generator added new data and the new data, because of the problem with ORM layer, Hibernate caused 5 more query executions in the database layer after each customer flow step. So it is a classical N +1 select ORM problem.

5.2.2.1 Setup

Stagemonitor was attached to the AIO core module as the application was deployed locally. Load generator was set up to simulated a registration process and a similar repeatable customer flow for 50 times, each time changing the state of the application and adding new data.

The source code of the AIO and the required dependencies for running AIO load generator are not included in the repository of this thesis [1]. Although, the readable load generator code for AIO is available. It reflects a most used segment of the whole application. Even the segment is about three dozen of REST services which require a lot of setups. It gives an overview of the scale of the AIO application and capability of the framework introduced in this thesis.

5.2.2.2 Detected performance problems

From the large application case studies three results are explained. First, the performance problem which was known before, secondly a newly found performance issue and for last a

performance problem which was as severe as first one but was not noticed because it did not affect the customer flow as much as the first one.

```
Processing HTTP request metrics. Data set size     50. HTTP request: Request-
id=00720  GET  /loanengine/rest/contracts/2558060
Issue diffAverage-> EXECUTION_TIME                  value       8,94 Request-
id=00720  GET  /loanengine/rest/contracts/2558060
Issue diffAverage-> CALLING_CONTEXT_TREE_SIZE       value   22140,04 Request-
id=00720  GET  /loanengine/rest/contracts/2558060
Issue diffAverage-> CALLING_CONTEXT_TREE_DEPTH      value       3,48 Request-
id=00720  GET  /loanengine/rest/contracts/2558060
Issue diffAverage-> DB_QUERY_COUNT                  value       5,08 Request-
id=00720  GET  /loanengine/rest/contracts/2558060
Issue diffAverage-> DB_EXECUTION_TIME               value       4,70 Request-
id=00720  GET  /loanengine/rest/contracts/2558060
```

Figure 12 Data analyzer output - AIO contract service

On Figure 12 is a segment from the full output of the analyzer which is available in Appendix 2. The segment represents REST service which retrieves AIO contract information based on contract identifier. As we can see every request executed due to additional data the application does on average in 5 extra DB queries, because of that the execution time also is on average 9 milliseconds greater than the previous request execution time.

```
Processing HTTP request metrics. Data set size   101. HTTP request: Request-
id=00810  GET  /loanengine/rest/contracts/2558060/account-statement

Issue diffAverage-> EXECUTION_TIME                value      1,79 Request-
id=00810  GET  /loanengine/rest/contracts/2558060/account-statement

Issue diffAverage-> DB_QUERY_COUNT                value      1,91 Request-
id=00810  GET  /loanengine/rest/contracts/2558060/account-statement

Issue diffAverage-> CALLING_CONTEXT_TREE_SIZE     value   8478,79 Request-
id=00810  GET  /loanengine/rest/contracts/2558060/account-statement

Issue diffAverage-> DB_EXECUTION_TIME             value      1,06 Request-
id=00810  GET  /loanengine/rest/contracts/2558060/account-statement

Issue diffAverage-> CALLING_CONTEXT_TREE_DEPTH    value      1,48 Request-
id=00810  GET  /loanengine/rest/contracts/2558060/account-statement
```

Figure 13 Data analyzer output - AIO account statement service

On Figure 13 is a segment from the full analyzer output. The segment represents AIO REST service which returns account statement by AIO contract identifier. The service is used in a separate window which is optionally openable during the customer flow steps. As we can see, due to additional data added by load generation method which is described in subtopic 4.4.1.1, every next request results in average of 1.06 SQL queries executed on the database layer. Same with execution time which is in average is 1.79 milliseconds longer because of application layer requires additional resources to set up and process the additional queries. For a customer with longer application use history, this could result in an increase in time around 500ms which is a serious issue.

```
Processing HTTP request metrics. Data set size     50. HTTP request:
Request-id=00730  GET
/loanengine/rest/contracts/2558060/invoices/open

Issue diffAverage-> CALLING_CONTEXT_TREE_SIZE       value  20723,76
Request-id=00730  GET
/loanengine/rest/contracts/2558060/invoices/open

Issue diffAverage-> CALLING_CONTEXT_TREE_DEPTH      value      2,80
Request-id=00730  GET
/loanengine/rest/contracts/2558060/invoices/open

Issue diffAverage-> EXECUTION_TIME                  value      9,12
Request-id=00730  GET
/loanengine/rest/contracts/2558060/invoices/open

Issue diffAverage-> DB_EXECUTION_TIME               value      5,56
Request-id=00730  GET
/loanengine/rest/contracts/2558060/invoices/open

Issue diffAverage-> DB_QUERY_COUNT                  value      4,88
Request-id=00730  GET
/loanengine/rest/contracts/2558060/invoices/open
```

Figure 14 Data analyzer output - AIO open amount service

On Figure 14 is a segment from the complete data analyzer output. The segment represents service which could be used to display the open amount of a particular contract. The service was used after the AIO contracts service and executed on the background. This explains that service did not cause load time problems and was not reported as a performance issue. As seen on the figure the execution time is 9,12 and additional database query count 4,88. A very similar results compared to data on Figure 12. A bit deeper investigation of the source code reveals commonly used service which iterated over all the cash flows of all the related invoices. So a similar root cause of the severe performance problem was detected.

The full list of data analyzer AIO results available in Appendix 2

6. Summary

The central goal of this thesis was to provide a solution which could be used to find and track performance issues in the application at the mature stages of the lifecycle. Along the process, research of available performance tools, related work and supporting technology was done. To validate the effectiveness of the solution application two distinct case studies were provided.

The current thesis introduces a framework for empirical performance evaluation for Java applications. The framework introduces a radical process of caching performance issue during the developer side testing phase of an application.

Two case studies were provided with the use of this framework. First case study with a smaller web framework sample application proved that the framework is competent to detect introduced performance problems in the development process. First case study suggests that the framework is well suited for empirical approach and could be integrated into a continuous integration process.

The second case study demonstrated that the framework is also usable for finding performance issues in a large web application (160K LOC). The case study identified already known performance issues in the large application which indicated that the new framework performs as expected. In addition, the framework provided metrics which could be used to evaluate the severity of a performance issue for example in terms of scalability.

In addition, the second case study also identified several new performance problems which were unknown until recent use of the framework. The problems were not as severe as the previously known ones and were located in seldom used services.

The framework is well suited for finding performance problems from the even very large application at late lifecycle.

With a first case study the frameworks provided means to validation and tracking of fixed and recently introduced performance issues. It makes the framework extremely useful as a component in continuous integration process.

The goals of the thesis are achieved. Tool is available as an open source project on Bitbucket. [1]

Kokkuvõte

Kesksek töö eesmärgiks oli välja tuua kasutatav lahendus, millega tuvastada ja jälgida jõudluse probleeme hilisemas elutsükklis olevates rakendustes. Protsessi käigus oli tehti uurimus olemasolevatest jõudluse seire vahendistes, akadeemilistest töödest ja seire vahendite aluseks olevast tehnoloogiast. Et valideerida lahenduse tõhusust, oli tehtud kaks uurimust.

Käesolev töö pakub välja raamistiku jõudluse empiiriliseks hindamiseks Java rakendustes. See raamistik tutvustab uudset protsessi rakenduse probleemide tuvastamiseks arendaja poolsel testimise etapil.

Seda raamistikku kasutades oli läbi viidud kaks uurimust. Esimene uurimus väiksema rakenduse peal tõestas, et see raamistik on kompetentne tuvastama jõudluse probleeme arendusprotsessi sees. See uuring näitab, et raamistik on sobilik empiiriliseks lähenemiseks ja võib olla integreeritud järjepideva koostamise protsessi.

Teine läbi viidud uurimus näitas, et raamistik on ka kasutatav suurema mastaabiga rakenduses (160 tuhat koodirida) jõudluse probleemide leidmiseks. Uurimuse käigus tuvastas raamistik suures rakenduses juba varem tuntud jõudluse vea, millest võib eeldada lähenemise tõhusust. Lisaks toob raamisik välja meetrilised väärtused, mille põhjal võib hinnata tuvastatud jõudluse probleemi ulatust näiteks rakenduse skaleeritavuse mõistes.

Teine uurimus tuvastas rakenduses probleeme, kuid seekord uusi, varem mitte tuntud jõudluse probleeme. Need jõudluse probleemid olid vähem tõsisemad või asusid harvemini ettetulevas kohas, mis selgitab, miks neid pole varem märgatud.

Töös tutvustatud raamistik on hästi sobiv jõudluse probleemide leidmiseks isegi suurema rakenduse korral, mis on hilisemas elutsükklis.

Esimese uurimuse korral näitas see raamistik viise jõudluse probleemide jälgimiseks ja hiljuti sisse viidud probleemi paranduste valideerimist. See teeb raamistikust kasuliku komponendi järjepideva juurutuse protsessis.

Töö eesmärgid said saavutatud ning loodud raamistik on avalikult saadaval koodihoidlas Bitbucket. [1]

References

[1] V. Reinok, "Theis bitbucket repository," Tallinn Univeristy of Tehnology, 12 05 2016. [Online]. Available: https://bitbucket.org/viktor_reinok_thesis_team/thesis/.

[2] I. Mägi, "Plumbr blog," Plumbr, 28 03 2016. [Online]. Available: https://plumbr.eu/blog/user-experience/performance-causing-users-abandon-site . [Accessed 06 05 2016].

[3] I. Mägi, "Plumbr blog," Plumbr, 04 05 2016. [Online]. Available: https://plumbr.eu/blog/user-experience/how-to-derive-business-value-from-performance-monitoring. [Accessed 06 05 2016].

[4] "Jenkins homepage," [Online]. Available: https://jenkins.io/. [Accessed 03 05 2016].

[5] „Jenkins performance plugin,“ [Võrgumaterjal]. Available: https://wiki.jenkins-ci.org/display/JENKINS/Performance+Plugin. [Kasutatud 04 05 2016].

[6] "JMeter homepage," [Online]. Available: http://jmeter.apache.org/ . [Accessed 5 05 2016].

[7] W. Binder, J. Hulaas and P. Moret, "Advanced Java Bytecode Instrumentation," in *University of Lugano*, Lugano, 2007.

[8] A. O. Anil Chawla, "A Generic Instrumentation Framework for Collecting Dynamic Information," *WERST Proceedings/ACM SIGSOFT SEN* , p. 1Volume 29 Number 5, 09 2004.

[9] W. B. A. V. P. M. Danilo Ansaloni, „Rapid Development of Extensible Profilers for the Java Virtual Machine with Aspect-Oriented Programming,“ *WOSP/SIPEW',* January 2010 .

[10] W. B. P. M. D. A. Alex Villazón, "Comprehensive aspect weaving for Java," *Science of Computer Programming*, pp. Volume 76, Issue 11, 1, November 2011.

[11] "Takipi," Takipi, [Online]. Available: https://www.takipi.com/. [Accessed 25 04 2016].

[12] R. Marvin, "PC Mag Digital Edition," 14 10 2015. [Online]. Available: http://www.pcmag.com/article2/0,2817,2492700,00.asp. [Accessed 18 04 2016].

[13] "Startup report," Funderbeam, [Online]. Available: https://www.funderbeam.com/startups/plumbr?ref=teleport. [Accessed 26 04 2016].

[14] "AppDynamics homepage," [Online]. Available: https://www.appdynamics.com/. [Accessed 16 04 2016].

[15] "NewRelic ducmentation page," [Online]. Available: https://docs.newrelic.com/docs/agents/java-agent/getting-started/new-relic-java. [Accessed 17 04 2016].

[16] "Dynatrace homepage - java," [Online]. Available: http://www.dynatrace.com/en/application-monitoring/technologies/java-monitoring/. [Accessed 17 04 2016].

[17] "Plumbr homepage," [Online]. Available: https://plumbr.eu/. [Accessed 17 04 2016].

[18] "XRebel homepage," ZeroTurnaround, [Online]. Available: https://zeroturnaround.com/software/xrebel/. [Accessed 18 04 2016].

[19] evernat, "Javamelody repository," [Online]. Available: https://github.com/javamelody/javamelody. [Accessed 18 04 2016].

[20] "Kibana homepage," Elasticsearch BV, [Online]. Available: https://www.elastic.co/products/kibana. [Accessed 06 05 2016].

[21] F. Barnsteiner, "Stagemonitor Github," iSYS Software GmbH, [Online]. Available: https://github.com/stagemonitor/stagemonitor/. [Accessed 26 04 2016].

[22] C. Y. H. J. Woonduk Kang, "Pinpoint Github," Naver, [Online]. Available: https://github.com/naver/pinpoint . [Accessed 26 04 2016].

[23] "Instrumentation," Wikipedia, 29 01 2015. [Online]. Available: https://en.wikipedia.org/wiki/Instrumentation_(computer_programming). [Accessed 20 03 2016].

[24] "Merriam Webster online dictionary," [Online]. Available: http://www.merriam-webster.com/dictionary/instrument . [Accessed 8 04 2016].

[25] "Instrumentation JavaDoc v5," Oracle, [Online]. Available: http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/instrument/Instrumentation.html. [Accessed 20 03 2016].

[26] "Soot homepage," [Online]. Available: https://sable.github.io/soot/. [Accessed 14 05 2016].

[27] "FindBugs homepage," FindBugs, [Online]. Available: http://findbugs.sourceforge.net/. [Accessed 20 03 2016].

[28] J. Aarniala, "Instrumenting Java bytecode," in *Seminar work for the Compilerscourse, pages 2,4* , 2005.

[29] "Hibernate documentation," [Online]. Available: http://agori.github.io/one_to_one_optional.html. [Accessed 21 03 2016].

[30] "Hibernate community documentation," [Online]. Available: https://docs.jboss.org/hibernate/orm/4.2/manual/en-US/html/ch20.html#bytecode-enhancement. [Accessed 21 03 2016].

[31] "OWASP Bytecode obfuscation," [Online]. Available: https://www.owasp.org/index.php/Bytecode_obfuscation. [Accessed 21 02 2016].

[32] "Threadlocal Javadoc 5," Oracle, [Online]. Available: https://docs.oracle.com/javase/1.5.0/docs/api/java/lang/ThreadLocal.html. [Accessed 28 04 2016].

[33] T. Parsons, "Logentries blog," logentries, 09 2014. [Online]. Available: https://blog.logentries.com/2014/09/how-to-trace-transactions-across-every-layer-of-your-distributed-software-stack/. [Accessed 4 04 2016].

[34] "Retroveaver homepage," [Online]. Available: http://retroweaver.sourceforge.net/ . [Accessed 30 03 2016].

[35] K. O'Hair, "The JVMPI Transition to JVMTI," Oracle, 1 July 2004. [Online]. Available: http://www.oracle.com/technetwork/articles/java/jvmpitransition-138768.html . [Accessed 11 04 2016].

[36] "ASM homepage," [Online]. Available: http://asm.ow2.org/. [Accessed 21 03 2016].

[37] E. Brundeton, "ASM 5.0 Documention," [Online]. Available: http://asm.ow2.org/asm50/javadoc/user/. [Accessed 1 February 2016].

[38] "Post build task - Jenkins plugin3," Jenkins, [Online]. Available: https://wiki.jenkins-ci.org/display/JENKINS/Post+build+task. [Accessed 07 05 2016].

[39] "Grinder load generator homepage," [Online]. Available: http://grinder.sourceforge.net/ . [Accessed 5 05 2016].

[40] "Jersey homepage," Oracle Corporation, [Online]. Available: https://jersey.java.net/. [Accessed 06 05 2016].

[41] "Elasticsearch homepage," [Online]. Available: https://www.elastic.co/. [Accessed 07 05 2016].

[42] "Petclinic repository," [Online]. Available: https://github.com/spring-projects/spring-petclinic . [Accessed 06 05 2016].

Appendices


```
{
   "took":49,
   "timed_out":false,
   "_shards":{
      "total":5,
      "successful":5,
      "failed":0
   },
   "hits":{
      "total":1,
      "max_score":6.9739127,
      "hits":[
         {
            "_index":"stagemonitor-requests-2016.05.04",
            "_type":"requests",
            "_id":"AVR7qFdP7e_ZliFNZw_o",
            "_score":6.9739127,
            "_source":{
               "bytesWritten":19553,
               "headers":{
                  "x-country":"EE",
                  "x-brand":"sving",
                  "request-name":"000_00770_0050",
                  "period-number":"0050",
                  "x-language":"et",
                  "request-id":"00770",
                  "accept":"text/html, image/jpeg, *; q=.2, */*; q=.2",
                  "host":"localhost:8080",
                  "content-type":"application/json",
                  "connection":"keep-alive",
                  "randomheader":"randomHeader",
                  "thread-id":"000",
                  "mesurement-id":"0000",
                  "user-agent":"Java/1.7.0_79"
               },
               "measurement_start":1462362719387,
               "instance":"loanengine",
               "method":"GET",
               "userAgent":{
                  "os":"JVM (Java)",
                  "osFamily":"JVM",
                  "osVersion":"1.7.0_79",
                  "browser":"Java",
                  "browserVersion":"1.7.0_79",
                  "type":"Library",
                  "device":"Other"
               },
               "containsCallTree":true,
               "sessionId":"3caaf2gwzsns1gqm0l1ui43fv",
               "error":false,
               "url":"/loanengine/rest/contracts/2558060/draw-selections",
               "executionTimeDb":492,
               "executionTime":1204,
               "@timestamp":"2016-05-04T15:05:09.341+0300",
               "application":"aio",
               "executionCountDb":361,
               "callStack":"...",
               "name":"GET /rest/contracts/2558060/draw-selections",
               "host":"MINDNOTE-014",
               "id":"7ed7bb67-d9f1-468b-85c0-6dc5ba71d89f",
               "executionTimeCpu":717,
               "uniqueVisitorId":"df889ae50f0ac88bf80a5e19c4b7a4957b118b25",
               "parameters":{

               },
               "statusCode":200,
               "status":"OK"
            }
         }
      ]
   }
}
```


Appendix 1 Stagemonitor request

Processing HTTP request metrics. Data set size 100. HTTP request: Request-id=12000 PUT /loanengine/rest/developer/contracts/allocate
Processing HTTP request metrics. Data set size 51. HTTP request: Request-id=00000 POST /loanengine/rest/authentication/banks_ee/dummy_bank_ee_id/confirm
Issue diffAverage-> CALLING_CONTEXT_TREE_SIZE value 3,94 Request-id=00000 POST /loanengine/rest/authentication/banks_ee/dummy_bank_ee_id/confirm
Processing HTTP request metrics. Data set size 51. HTTP request: Request-id=00100 GET /loanengine/rest/authentication/banks_ee/
Processing HTTP request metrics. Data set size 51. HTTP request: Request-id=00110 POST /loanengine/rest/authentication/banks_ee/dummy_bank_ee_id
Issue diffAverage-> CALLING_CONTEXT_TREE_SIZE value 4,29 Request-id=00110 POST /loanengine/rest/authentication/banks_ee/dummy_bank_ee_id
Processing HTTP request metrics. Data set size 51. HTTP request: Request-id=11000 DELETE /loanengine/rest/authentication
Processing HTTP request metrics. Data set size 50. HTTP request: Request-id=00710 GET /loanengine/rest/authentication
Issue diffAverage-> CALLING_CONTEXT_TREE_SIZE value 17,30 Request-id=00710 GET /loanengine/rest/authentication
Processing HTTP request metrics. Data set size 50. HTTP request: Request-id=00720 GET /loanengine/rest/contracts/2558060
Issue diffAverage-> EXECUTION_TIME value 8,94 Request-id=00720 GET /loanengine/rest/contracts/2558060
Issue diffAverage-> CALLING_CONTEXT_TREE_SIZE value 22140,04 Request-id=00720 GET /loanengine/rest/contracts/2558060
Issue diffAverage-> CALLING_CONTEXT_TREE_DEPTH value 3,48 Request-id=00720 GET /loanengine/rest/contracts/2558060
Issue diffAverage-> DB_QUERY_COUNT value 5,08 Request-id=00720 GET /loanengine/rest/contracts/2558060
Issue diffAverage-> DB_EXECUTION_TIME value 4,70 Request-id=00720 GET /loanengine/rest/contracts/2558060
Processing HTTP request metrics. Data set size 50. HTTP request: Request-id=00730 GET /loanengine/rest/contracts/2558060/invoices/open
Issue diffAverage-> CALLING_CONTEXT_TREE_SIZE value 20723,76 Request-id=00730 GET /loanengine/rest/contracts/2558060/invoices/open
Issue diffAverage-> CALLING_CONTEXT_TREE_DEPTH value 2,80 Request-id=00730 GET /loanengine/rest/contracts/2558060/invoices/open
Issue diffAverage-> EXECUTION_TIME value 9,12 Request-id=00730 GET /loanengine/rest/contracts/2558060/invoices/open
Issue diffAverage-> DB_EXECUTION_TIME value 5,56 Request-id=00730 GET /loanengine/rest/contracts/2558060/invoices/open
Issue diffAverage-> DB_QUERY_COUNT value 4,88 Request-id=00730 GET /loanengine/rest/contracts/2558060/invoices/open
Processing HTTP request metrics. Data set size 50. HTTP request: Request-id=00740 GET /loanengine/rest/credit-application/loanissuers
Processing HTTP request metrics. Data set size 50. HTTP request: Request-id=00750 GET /loanengine/rest/contracts/2558060/invoices/outstanding
Issue diffAverage-> EXECUTION_TIME value 8,44 Request-id=00750 GET /loanengine/rest/contracts/2558060/invoices/outstanding
Issue diffAverage-> DB_QUERY_COUNT value 4,28 Request-id=00750 GET /loanengine/rest/contracts/2558060/invoices/outstanding
Issue diffAverage-> CALLING_CONTEXT_TREE_DEPTH value 2,28 Request-id=00750 GET /loanengine/rest/contracts/2558060/invoices/outstanding
Issue diffAverage-> CALLING_CONTEXT_TREE_SIZE value 17908,80 Request-id=00750 GET /loanengine/rest/contracts/2558060/invoices/outstanding
Issue diffAverage-> DB_EXECUTION_TIME value 4,90 Request-id=00750 GET /loanengine/rest/contracts/2558060/invoices/outstanding
Processing HTTP request metrics. Data set size 50. HTTP request: Request-id=00760 GET /loanengine/rest/contracts/2558060/extra-services
Issue diffAverage-> CALLING_CONTEXT_TREE_DEPTH value 0,36 Request-id=00760 GET /loanengine/rest/contracts/2558060/extra-services
Issue diffAverage-> CALLING_CONTEXT_TREE_SIZE value 3461,48 Request-id=00760 GET /loanengine/rest/contracts/2558060/extra-services
Issue diffAverage-> DB_QUERY_COUNT value 0,80 Request-id=00760 GET /loanengine/rest/contracts/2558060/extra-services
Processing HTTP request metrics. Data set size 50. HTTP request: Request-id=00770 GET /loanengine/rest/contracts/2558060/draw-selections
Issue diffAverage-> DB_EXECUTION_TIME value 3,72 Request-id=00770 GET /loanengine/rest/contracts/2558060/draw-selections
Issue diffAverage-> EXECUTION_TIME value 5,34 Request-id=00770 GET /loanengine/rest/contracts/2558060/draw-selections
Issue diffAverage-> DB_QUERY_COUNT value 4,44 Request-id=00770 GET /loanengine/rest/contracts/2558060/draw-selections
Issue diffAverage-> CALLING_CONTEXT_TREE_DEPTH value 3,20 Request-id=00770 GET /loanengine/rest/contracts/2558060/draw-selections
Issue diffAverage-> CALLING_CONTEXT_TREE_SIZE value 19921,26 Request-id=00770 GET /loanengine/rest/contracts/2558060/draw-selections
Processing HTTP request metrics. Data set size 50. HTTP request: Request-id=00780 GET /loanengine/rest/products/CREDIT_LINE
Processing HTTP request metrics. Data set size 50. HTTP request: Request-id=00810 GET /loanengine/rest/contracts/2558060/account-statement
Issue diffAverage-> EXECUTION_TIME value 4,44 Request-id=00810 GET /loanengine/rest/contracts/2558060/account-statement
Issue diffAverage-> DB_EXECUTION_TIME value 2,16 Request-id=00810 GET /loanengine/rest/contracts/2558060/account-statement
Issue diffAverage-> CALLING_CONTEXT_TREE_DEPTH value 1,32 Request-id=00810 GET /loanengine/rest/contracts/2558060/account-statement
Issue diffAverage-> DB_QUERY_COUNT value 3,32 Request-id=00810 GET /loanengine/rest/contracts/2558060/account-statement
Issue diffAverage-> CALLING_CONTEXT_TREE_SIZE value 13322,76 Request-id=00810 GET /loanengine/rest/contracts/2558060/account-statement
Processing HTTP request metrics. Data set size 50. HTTP request: Request-id=00910 PUT /loanengine/rest/contracts/2558060/draw
Issue diffAverage-> CALLING_CONTEXT_TREE_DEPTH value 3,26 Request-id=00910 PUT /loanengine/rest/contracts/2558060/draw
Issue diffAverage-> EXECUTION_TIME value 8,96 Request-id=00910 PUT /loanengine/rest/contracts/2558060/draw
Issue diffAverage-> DB_EXECUTION_TIME value 4,80 Request-id=00910 PUT /loanengine/rest/contracts/2558060/draw
Issue diffAverage-> CALLING_CONTEXT_TREE_SIZE value 22276,90 Request-id=00910 PUT /loanengine/rest/contracts/2558060/draw
Issue diffAverage-> DB_QUERY_COUNT value 5,12 Request-id=00910 PUT /loanengine/rest/contracts/2558060/draw
Processing HTTP request metrics. Data set size 50. HTTP request: Request-id=00920 GET /loanengine/rest/contracts/2558060
Issue diffAverage-> CALLING_CONTEXT_TREE_DEPTH value 3,28 Request-id=00920 GET /loanengine/rest/contracts/2558060
Issue diffAverage-> CALLING_CONTEXT_TREE_SIZE value 21724,96 Request-id=00920 GET /loanengine/rest/contracts/2558060
Issue diffAverage-> DB_QUERY_COUNT value 5,00 Request-id=00920 GET /loanengine/rest/contracts/2558060
Issue diffAverage-> EXECUTION_TIME value 10,56 Request-id=00920 GET /loanengine/rest/contracts/2558060
Issue diffAverage-> DB_EXECUTION_TIME value 4,44 Request-id=00920 GET /loanengine/rest/contracts/2558060
Processing HTTP request metrics. Data set size 50. HTTP request: Request-id=00930 GET /loanengine/rest/contracts/2558060/account-statement
Issue diffAverage-> DB_EXECUTION_TIME value 4,04 Request-id=00930 GET /loanengine/rest/contracts/2558060/account-statement
Issue diffAverage-> DB_QUERY_COUNT value 4,62 Request-id=00930 GET /loanengine/rest/contracts/2558060/account-statement
Issue diffAverage-> CALLING_CONTEXT_TREE_SIZE value 20335,02 Request-id=00930 GET /loanengine/rest/contracts/2558060/account-statement
Issue diffAverage-> CALLING_CONTEXT_TREE_DEPTH value 3,32 Request-id=00930 GET /loanengine/rest/contracts/2558060/account-statement
Issue diffAverage-> EXECUTION_TIME value 7,58 Request-id=00930 GET /loanengine/rest/contracts/2558060/account-statement
Processing HTTP request metrics. Data set size 50. HTTP request: Request-id=00940 GET /loanengine/rest/products/CREDIT_LINE
Processing HTTP request metrics. Data set size 50. HTTP request: Request-id=00950 GET /loanengine/rest/contracts/2558060/extra-services
Issue diffAverage-> CALLING_CONTEXT_TREE_SIZE value 3048,22 Request-id=00950 GET /loanengine/rest/contracts/2558060/extra-services
Issue diffAverage-> CALLING_CONTEXT_TREE_DEPTH value 0,26 Request-id=00950 GET /loanengine/rest/contracts/2558060/extra-services
Issue diffAverage-> DB_QUERY_COUNT value 0,70 Request-id=00950 GET /loanengine/rest/contracts/2558060/extra-services

Appendix 2 AIO data analyzer output