The goal of CloudScale is to aid service providers in analysing, predicting and resolving scalability issues, i.e., support scalable service engineering. The project extends existing and develops new solutions that support the handling of scalability problems of software-based services.

In this WP2 deliverable, we summarize our results of the evolution support in the CloudScale method. The evolution support focuses on migrating the existing system to the cloud computing environments. The results of the evolution support include:

- Overview of the evolution support.
- Extractor, especially its first validation
- The Static Spotter
- The Dynamic Spotter

### Dissemination level

<table>
<thead>
<tr>
<th>PU</th>
<th>Public</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CO</td>
<td>Confidential, only for members of the consortium (including Commission Services)</td>
<td>x</td>
</tr>
</tbody>
</table>

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 317704.
CloudScale consortium

CloudScale (Contract No. FP7-317704) is a Collaboration Project (STREP) within the 7th Framework Programme, Call 8, Priority 1.2 (Cloud Computing). The consortium members are:

SINTEF ICT
(SINTEF, Norway)
NO-7465 Trondheim
Norway
www.sintef.com

Project manager: Richard T. Sanders
richard.sanders@sintef.no
+47 930 58 954
Technical manager: Gunnar Brataas
gunnar.brataas@sintef.no
+47 472 36 938

SAP Research, CEC
Karlsruhe
(SAP, Germany)
69190 Walldorf, Germany
www.sap.com/research

Contact: Roozbeh Farahbod
roozbeh.farahbod@sap.com

Ericsson Nikola Tesla
(ENT, Croatia)
Krapinska 42
10000 Zagreb, Croatia
www.ericsson.com/hr

Contact: Darko Huljenic
darko.huljenic@ericsson.com

XLAB, Slovenia
Pot za Brdom 100
1000 Ljubljana, Slovenia
www.xlab.si

Contact: Jure Polutnik
jure.polutnik@xlab.si

University of Paderborn
(UPB, Germany)
Zukunftsmile 1
33102 Paderborn, Germany
www.uni-paderborn.de

Contact: Steffen Becker
steffen.becker@uni-paderborn.de

Technische Universität Chemnitz (TUC, Germany)
Straße der Nationen 62
09107 Chemnitz, Germany
www.tu-chemnitz.de

Contact: Steffen Becker
steffen.becker@informatik.tu-chemnitz.de

1 UPB withdrew from the Consortium in October 2014, as the research team there relocated to TUC. TUC was not a partner before October 2014.
Table of contents

CloudScale consortium ................................................................................................................... ii
Table of contents .......................................................................................................................... iii
List of figures .................................................................................................................................. iv
List of Tables ................................................................................................................................... iv
1 Introduction .................................................................................................................................. 6
   1.1 CloudScale motivation and background ........................................................................ 6
   1.2 Relationships with other deliverables ............................................................................ 7
   1.3 Relation to First Version of This Deliverable ............................................................... 7
   1.4 Relation to Final Version of This Deliverable ............................................................... 7
   1.5 Contributors .................................................................................................................... 7
   1.6 Acronyms and abbreviations ......................................................................................... 8
2 Overview of evolution support .................................................................................................. 9
3 Documenting Scalability Anti-patterns/HowNotTos ................................................................. 10
   3.1 Anti-pattern Catalogue .................................................................................................... 10
   3.2 Anti-pattern Example “One Lane Bridge” ....................................................................... 10
4 The Static Spotter ....................................................................................................................... 12
   4.1 Methodology .................................................................................................................. 12
   4.2 Implementation details .................................................................................................... 13
   4.3 Test results ...................................................................................................................... 13
   4.4 Integration plan of Static and Dynamic Spotters for Y3 ................................................ 13
5 Scalability analysis by systematic experimentation ................................................................. 15
   5.1 Overview ........................................................................................................................ 15
   5.2 Performance Problem Hierarchy .................................................................................... 15
   5.3 Detecting Performance Problems .................................................................................... 16
      5.3.1 Detection Strategy .................................................................................................... 17
      5.3.2 Deriving Heuristics ................................................................................................. 17
   5.4 Software Bottlenecks ....................................................................................................... 17
      5.4.1 Directed Growth Strategy ...................................................................................... 18
      5.4.2 Time Windows Strategy ......................................................................................... 18
      5.4.3 One-Lane Bridge .................................................................................................. 18
   5.5 Inter-component Communication Anti-patterns .............................................................. 19
      5.5.1 Excessive Messaging ............................................................................................... 19
      5.5.2 Communication Blob ............................................................................................. 20
      5.5.3 Empty Semi Trucks (EST) .................................................................................... 20
      5.5.4 The Stifle ................................................................................................................ 20
   5.6 Summary / Discussion ........................................................................................................ 20
      5.6.1 Availability of Usage Profiles ................................................................................ 21
      5.6.2 The Nature of the Problems ................................................................................... 21
      5.6.3 Usage of Heuristics ............................................................................................... 21
6 The Extractor .............................................................................................................................. 22
   6.1 ENT use-case ................................................................................................................... 22
   6.2 Test and validation ............................................................................................................ 22
7 Conclusions and future work ................................................................................................... 25
References ....................................................................................................................................... 26
List of figures

Figure 1: The results of CloudScale ................................................................. 6
Figure 2: CloudScale method overview ............................................................ 9
Figure 3: Overview of a pattern catalogue ...................................................... 12
Figure 4: AcquireReleasePair ........................................................................ 12
Figure 5: Results from the Static Spotter ....................................................... 13
Figure 6: Symptoms of Known Performance Problems .................................. 15
Figure 7: Performance Anti-patterns of Varying Response Time .................. 16
Figure 8: Communication Performance Anti-patterns ................................... 19

List of Tables

Table 1: Specification of the machines used for testing user installation of the Extractor tool 22
Table 2: The summary of the Extractor developer installation test results ............ 23
Table 3: Parameters and results of the composition/merging tests ................... 23
Executive summary

In this WP2 deliverable, we summarize our results of the evolution support in the CloudScale method. The evolution support focuses on migrating the existing system to the cloud computing environments. The results of the evolution support with focus on Y2 include:

- **Overview of the evolution support.** In this chapter, we describe how we support software evolution within the CloudScale method and its progress. There are three tools related to Evolution Support: The Extractor extracts software architecture from source codes and visualizes with ScaleDL models. The Dynamic Spotter finds components responsible for hindering scalability by systematically executing the implemented system, while the Static Spotter inspects the software architecture and the system’s source code.

- **Modelling and analysing scalability anti-patterns (HowNotTo, bad practices).** In this chapter, we document an example anti-pattern, which we formalised in the Static Spotter in Year 2.

- **The Static Spotter.** In this chapter, we report on the methodology of the Static Spotter and the test results by applying ENT use cases.

- **The Dynamic Spotter.** In this chapter, we refine our tool of the Spotting by Measuring method and its complementing tool, the Dynamic Spotter.

- **The Extractor.** We report the test results by applying ENT use cases on the Extractor.
1 Introduction

1.1 CloudScale motivation and background

Cloud providers theoretically offer their customers unlimited resources for their applications on an on-demand basis. However, scalability is not only determined by the available resources, but also by how the control and data flow of the application or service is designed and implemented. Implementations that do not consider their effects can either lead to low performance (under-provisioning, resulting in high response times or low throughput) or high costs (over-provisioning, caused by low utilisation of resources).

CloudScale provides an engineering approach for building scalable cloud applications and services. Our objectives are to:

1. Make cloud systems scalable by design so that they can exploit the elasticity of the cloud, as well as maintaining and also improving scalability during system evolution. At the same time, a minimum amount of computational resources shall be used.

2. Enable analysis of scalability of basic and composed services in the cloud.

3. Ensure industrial relevance and uptake of the CloudScale results so that scalability becomes less of a problem for cloud systems.

CloudScale enables the modelling of design alternatives and the analysis of their effect on scalability and cost. Best practices for scalability further guide the design process.

The engineering approach for scalable applications and services will enable small and medium enterprises as well as large players to fully benefit from the cloud paradigm by building scalable and cost-efficient applications and services based on state-of-the-art cloud technology. Furthermore, the engineering approach reduces risks as well as costs for companies newly entering the cloud market.
1.2 Relationships with other deliverables

The results presented in this document relates on the following deliverables:

- **D1.2** – Design support covers the CloudScale method, ScaleDL (Architectural Templates, Usage Evolution and Overview) and design patterns that are important for integration of the tools and knowledge in the CloudScale Environment.

- **D3.2** – Second version of Integrated Tools: Presenting the architecture of the tools, which are used in the CloudScale Method. This deliverable also presents the ScaleDL Overview which focuses on deployment.

- **D4.2** – Requirements and validation, second version: which gives the requirements for ScaleDL Usage Evolution, ScaleDL Architectural Templates and also for the Analyser. CloudScale Method usage effort estimations baseline is defined based on the initial validation results for this deliverable.

- **D5.2** – Second version of Showcase: Presents the CloudStore showcase, which served as test case for the tools developed to support evolution.

1.3 Relation to First Version of This Deliverable

Compared to the first deliverable from WP2, D2.1, this second deliverable provides the following main changes and extensions to these chapters:

- Overview of Evolution Support: We have further completed the evolution support as outlined in the first version of this deliverable.

- Modelling and analysing scalability anti-patterns: We have formalized one more anti-pattern this year and add one more field to the template, Linkage with good practices.

- The Static Spotter: This section is new. We develop the tool Static Spotter this year. We explain the methodology in Section 4.1 and report its test with ENT use cases in Section 4.2. We also plan an integration of both spotters, the Dynamic Spotter and Static Spotter.

- Scalability analysis by systematic experimentation: We further refined our tool the Dynamic Spotter and add more detection strategies in Section 5.5.

- The Extractor: The methodology stays the same as in D2.1. This year we evaluated the Extractor with ENT use cases and we report the results in Section 6.2.

1.4 Relation to Final Version of This Deliverable

This deliverable is the second of three iterations and now presents all the relevant tools for evolution support, the Extractor, the Static and the Dynamic Spotter:

- Overview of Evolution Support: We will complete the methodology for evolution support

- Modelling and analysing scalability anti-patterns: More anti-patterns will be formalized so that we will have a rather complete catalogue.

- The Static Spotter: We will further develop and evaluate the Static Spotter. The Static Spotter will be able to detect more anti-patterns. It will provide input to the Dynamic Spotter as outlined in Section 4.

- Scalability analysis by systematic experimentation: We will further refine our tool the Dynamic Spotter and add more detection strategies and integrate the Dynamic Spotter with the Static Spotter.

- The Extractor: We will fix all the known bugs and further develop the Extractor due to the need of different requirements from the Static Spotter.

1.5 Contributors

The following partners have contributed to this deliverable:
1.6 Acronyms and abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCM</td>
<td>Palladio Component Model</td>
</tr>
<tr>
<td>PINOT</td>
<td>Pattern INference recOvery Tool</td>
</tr>
<tr>
<td>PoC</td>
<td>Proof of Concept</td>
</tr>
<tr>
<td>SoMoX</td>
<td>Software MOdel eXtractor</td>
</tr>
<tr>
<td>UPB</td>
<td>TUC</td>
</tr>
</tbody>
</table>
2 Overview of evolution support

In this chapter, we briefly describe the overview of the evolution support progress in the CloudScale method.

The overview of CloudScale method is shown in Figure 2. WP2 is mainly involved in the evolution scenario, highlighted via three green arrows. This evolution scenario covers methods and tools to assist software engineers to evolve an existing software system to satisfy scalability requirements in cloud computing environments.

In Year 1, we developed the Extractor to reconstruct partial ScaleDL models from an existing software system (evolution scenario step (1)).

In Year 2, we developed the Static Spotter that could detect scalability anti-pattern candidates from the partial ScaleDL models (evolution scenario step (2)). Now software engineers can pin down potential factors, which affect the scalability of the system.

Afterwards, software engineers can use the refined Dynamic Spotter (coming with Year 3) to locate the actual factors (evolution scenario step (3)). The refined Dynamic Spotter now covers more anti-pattern, with particular focus on message exchange and has been released as open source project.
3 Documenting Scalability Anti-patterns/HowNotTos

In Year 1, we presented how we formalize a scalability anti-pattern. In Year 2, we present more scalability anti-patterns in our scalability anti-pattern catalogue available from the CloudScale Wiki [13].

Meanwhile, we differentiate on performance anti-pattern and scalability anti-pattern. Performance anti-patterns are the classical issues, which hinder a system to perform as expected under a given workload. Scalability anti-patterns are those anti-patterns, which prevent the system from scaling when the system has to face higher workloads.

3.1 Anti-pattern Catalogue

In Year 1, we started to build a catalogue of scalability anti-patterns. In Year 2, we have documented more anti-patterns in the catalogue. The catalogue can be found in our wiki. Several patterns have been formalised and implemented in our tools, the Static Spotter and the Dynamic Spotter. In this section we will give an example of a pattern description, which is used in both, the Static Spotter (c.f. Section 4) and the Dynamic Spotter (c.f. Section 5). Finally, in our pattern catalogue in the CloudScale Wiki, we have introduced a new field to document the linkage between bad and good practices. As anti-patterns are labelled as bad practices, our HowTos in the CloudScale Wiki sometimes provide potential solutions to the problems.

Below we provide an example of an scalability anti-pattern. Additional scalability anti-patterns are available on the wiki.

3.2 Anti-pattern Example “One Lane Bridge”

Name: One Lane Bridge

Also known as: Single Lane Bridge

Example: The one lane bridge situation originates from New Zealand. In New Zealand’s highway network, there are one lane bridges and these one lane bridges are even shared with trains. As the name indicates, only one direction of cars or trains are allowed to pass the one lane bridge at a time.

As for software, one typical analogy to One Lane Bridge is that a point where only one process may continue to execute concurrently and all the other processes have to wait. For example, when software comes to execute synchronized blocks, the other have to wait until it finishes.

Problem: The problem with a One Lane Bridge is that traffic may only travel in one direction at a time, and if there are multiple lanes of traffic all moving in parallel, they must merge and proceed across the bridge, one vehicle at a time. This increases the time required to get a given number of vehicles across the bridge and can also cause long backups.

In software One Lane Bridge is typically a point where one, or only a few, processes may continue to execute concurrently and all the other processes have to wait. One frequent example occurs when applications access a database. In order to prevent data from being corrupted, a lock is being applied which makes sure only one process may update the database at a time. It may also occur when a set of processes make a synchronous call to another process. For example, in an online book store application, a customer changes several items quantities in his shopping cart. That would trigger a series of update operations. These update operations will behave like One Lane Bridge. Only one update operates at a time; all others have to wait.

Solution: There are solutions to One Lane Bridge. One is to build a multi-lane bridge, one is to build multiple one lane bridge. In software, the solution is similar. Take database as an example. We may reconstruct the data structure so that multiple update operations could be performed at the same time. Or we may have multiple database instances and use an algorithm to make sure their data are not conflicting with each other.
Detection: In Java, it is common practice to use synchronized blocks or synchronized methods to make sure only one process is being executed at a time. Therefore, we could locate the anti-pattern via detecting synchronized blocks or synchronized methods.

Variants: not known do far

Consequences: the anti-pattern potentially becomes a barrier to satisfy responsiveness and scalability requirements.

See also: [C. Smith]
4 The Static Spotter

In this section we describe the Static Spotter, its methodology and the test results of our use-cases. The Static Spotter is a reverse engineering tool based on Reclipse for the automatic detection of so called search patterns which then are interpreted as potential scalability anti-patterns. Search patterns formalise certain source code structures, which may appear in existing code, e.g., a block of Java statements encapsulated in a synchronized block. Such a structure becomes a scalability anti-pattern if the synchronized block turns out to become a One Lane Bridge scalability anti-pattern in case of increased workload.

Here we use the Static Spotter to detect scalability anti-pattern candidates so that during the evolution phase, software engineers could locate potential or dormant scalability issues with the help of the Static Spotter.

4.1 Methodology

This section gives an overview of the Static Spotter’s methodology. The Static Spotter takes partial ScaleDL models as an input. The partial ScaleDL models could be the ones generated by the Extractor. The output of the Static Spotter is a ranked list of scalability anti-pattern candidates.

The Static Spotter does two things: first parse the code and models and then look for search patterns and interpret them as scalability pattern candidates. There is a pre-configuring step before parsing and searching. That is to set up the search pattern catalogues. In this catalogues, graph patterns are used to formalise the search patterns. Figure 3 shows a catalogue used to detect the scalability anti-pattern One Lane Bridge. Since synchronized blocks and synchronized methods are potential One Lane Bridge candidates Figure 4 depicts an AcquireReleasePair pattern (which is what the Extractor extracts from a synchronized block).

![Pattern Specifications contained in this catalogue](image)

**Figure 3: Overview of a pattern catalogue**

![AcquireReleasePair](image)

**Figure 4: AcquireReleasePair**
A static analysis parses partial ScaleDL models and searches for instances of the defined search patterns. The result of the static analysis is a set of code fragments that comply with the search pattern specifications. We then interpret some of these search patterns as potential scalability anti-patterns (while others just form helper search patterns to structure the search pattern catalogue). The tool gives for each scalability anti-pattern also a relevance ranking which tells the user the likelihood that the found scalability anti-pattern indeed is one.

**Figure 5: Results from the Static Spotter**

In Figure 5 you see example results after running the Static Spotter. In these results, the Static Spotter finds two instances of AcquireReleasePair and five instances of SynchronizedMethod. The 100% ranking means the Static Spotter would classify the candidates as certain scalability anti-patterns.

### 4.2 Implementation details

In order to implement the Static Spotter, we have spent effort to modernise the Reclipse tool initially developed as background by UPB. This tool has been developed to detect search pattern in source codes of software systems. Its initial use case was to look for undocumented design patterns in the code. We have evolved it, ported it to the latest Eclipse version and then extended it, so that it is now able to spot scalability anti-patterns. It does so by looking for predefined search patterns in the code, which can be modelled in a language provided by the tool. The latter makes adding further search pattern very easy so that the higher efforts for extending this language pays of as soon as we formalise further patterns in the last year.

In addition we had to extend the Extractor so that it extracts additional elements from the source code, e.g., it converts synchronized Java blocks into actions which are surrounded by Acquire-/Release-Pairs now.

### 4.3 Test results

We test the Static Spotter with the ENT use-cases. The Static Spotter successfully detects in this use case synchronized blocks and methods. During the tests, we notice that when the use cases contain a thousand classes or more, the Static Spotter takes long time to analyse. However, it should not affect the usage of the Static Spotter because while evolving software, it is acceptable to wait for analysis results. However, sometimes the Static Spotter fails because of a lack of memory. This is an inherent issue when dealing with large code bases which might not fit into memory for the search pattern detection step.

### 4.4 Integration plan of Static and Dynamic Spotters for Y3

In year 2 we made a conceptual plan to integrate the Static and Dynamic Spotters. The Static Spotter could find the potential scalability anti-pattern candidates, while the Dynamic Spotter could analyse whether these candidates are the actual affecting factors hindering the scalability requirements. To streamline the process, the Static Spotter provides a list of the potential candidates and then the Dynamic Spotter uses the candidate list to analyse.

This means we add another step after the initial pattern detection. In this second step, a dynamic analysis is used to confirm or reject the candidates depending on their runtime behaviour. The candidates’ expected behaviour is described with formal behavioural patterns based on UML 2.0 sequence diagrams. Each behavioural pattern corresponds to a structural pattern and references its
elements, e.g. object types and methods. After detecting the pattern candidates in the static analysis step, the software system under analysis is executed, and the candidates’ behaviour is traced. Note that not the complete program behaviour is traced but only that of the candidates. This drastically reduces the search space for the dynamic analysis. A number of traces is generated for each candidate. The traces are then compared to the behavioural patterns and it is evaluated how many of a candidate’s traces conform to the pattern. Based on the number of conform traces, software engineers can decide if a candidate is an actual pattern instance (i.e. a true positive) or a false positive.
5 Scalability analysis by systematic experimentation

This section will introduce CloudScale’s approach to analysing scalability issues through systematic measurements. We will first introduce the idea of the Spotting by Measuring method and its complementing tool, the Dynamic Spotter. We will then present the core ideas and the main elements of the method and the Dynamic Spotter in the rest of this section. More details about the tool, its implementation and integration into the CloudScale environment is provided in Deliverable D3.2.

5.1 Overview

The performance of an application is highly visible to end-users and thus crucial for its success. Response times, throughput, and resource consumption affect conversion rates, user satisfaction, and operational costs. However, performance and scalability problems are usually difficult to detect and even harder to reproduce. Low performance as a result of increasing work or load can have various causes in an application architecture, implementation, or deployment environment. Without sufficient expertise, it is hard to identify the actual cause of a problem. Software engineers need to know typical performance problems that can occur in their application. For each problem, they must know where and how to measure in order to get the necessary data without distorting measurements. In many cases, the necessary performance metrics cannot be collected. These will lead to incomplete and noisy measurement data which in turn make it even harder to draw the right conclusions.

The Dynamic Spotter tool as part of the CloudScale toolset, applies a novel Performance Problem Diagnostics (PPD) approach [2] that automatically identifies performance problems in an application and diagnoses their root causes. Once software engineers specified a usage profile for their application and setup a test system, the Dynamic Spotter can automatically search for known performance and scalability problems. Since the Dynamic Spotter encapsulates knowledge about typical performance problems and anti-patterns, only little performance engineering expertise is required for its usage. The Dynamic Spotter combines search techniques that narrow down the scope of the problem based on a decision tree with systematic experiments. The combination of both allows efficiently uncovering performance and scalability problems and their root causes that are otherwise hard to tackle.

5.2 Scalability Problem Hierarchy

We structure known performance and scalability problems in a hierarchy of symptoms (which are observable at the system border), problem (which describe typical performance & scalability problems within a software systems), and root causes (part that causes the problem). Figure 5.1 shows an excerpt of the hierarchical structure of performance problems. An extended performance problem hierarchy for a large set of the performance problems known in literature will be made available later in the project.

![Figure 6: Symptoms of Known Performance Problems](image)

The hierarchy is structured in categories, symptoms, problems, and root causes. The category Occurrences of High Response Times in Figure 6 groups common symptoms for the performance problems High Overhead, Varying Response Times, Unbalanced Processing, and Dispensable.
Computations. Symptoms represent the starting point for the performance problem diagnostics. They combine common characteristics of a set of performance problems. Each symptom is refined by more specific performance problems that further limit the set of possible root causes.

Figure 5 shows the problem hierarchy for Varying Response Times. We identified the anti-patterns The Ramp and Traffic Jam as potential causes of Varying Response Times. The Ramp occurs if response times of an application increase during operation. For example, a request to an online shop takes 10 ms when the store application has been started. After a couple of hours of operation the same request takes more than one second. Such behaviour can, for example, occur if the application contains Dormant References, i.e., the memory consumption of the application is growing over time. The root cause is Specific Data Structures which are growing during operation or which are not properly disposed. Another cause of Varying Response Times is the Traffic Jam anti-pattern. A Traffic Jam occurs if many concurrent threads or processes are waiting for the same shared resources. These can either be passive resources (like semaphores or mutexes) or active resources (like CPU or hard disk). In the first case, we have a typical One Lane Bridge whose critical resource needs to be identified. We focus on Synchronization Points (indicated by semaphores and synchronized methods), Database Locks, and Pools as potential root causes. In the case of limited physical resources, the root cause can only be a specific Bottleneck Resource.

Even though the presented hierarchy is not all-encompassing, it is extensible allowing for integration of further performance and scalability problems, symptoms, and root causes.

5.3 Detecting Scalability Problems

To detect scalability problems and to identify their root causes, the hierarchy introduced above serves as a decision tree that structures the search. Starting from the root nodes (representing symptoms of the problems), our algorithm looks for more and more specific symptoms and finally root causes. For example, symptoms require top-level metrics such as end-to-end response time or CPU utilization. If a certain symptom has been found, the algorithm systematically investigates its associated performance problems. For each problem (and root cause), we repeat the same process. With each step the problem becomes more specific and requires a more fine-grained instrumentation of the system under test. For each symptom, performance and scalability problem and root cause, we require a detection strategy. In this section we describe what detection strategies are and how they are used.
5.3.1 Detection Strategy

Basically, a detection strategy is a series of experiments that are executed against a system under test (SUT). Each detection strategy addresses a single performance problem or root cause. It is defined by

- **Workload variation**: an independent workload parameter to be varied from one experiment to the next.
- **Observed metrics**: performance metrics to be collected during each experiment defined by instrumentation rules (e.g., end-to-end response times or waiting times).
- **Analysis strategy**: analysis of measurement data to decide about the presence of a performance problem.

The Dynamic Spotter uses systematic experimentation to observe the effect of changes in the workload on the performance of the system under test. Such dependencies can indicate the existence of quality problems or can confirm a particular root cause. For example, it can observe changes of end-to-end response times with respect to the number of users. If the variance of response times increases disproportionately with the number of users, the Varying Response Times are an indication for a Traffic Jam or The Ramp. On a lower level, we can observe the waiting time of threads at a synchronization point. If the waiting times increase significantly with the number of users, the Synchronization Point is a potential root cause for a One Lane Bridge. While the first example decides if potential problems exist in the SUT as a whole, the latter identifies the root cause of a scalability problem.

All detection strategies are defined once and can then be executed against various applications fully automatically. To achieve this, we define detection strategies so that they can be applied to a class of applications (e.g., Java-based enterprise applications). Each detection strategy encapsulates heuristics for the identification of a particular problem. We define the required rules for dynamic instrumentation and workload variation as well as analysis methods (heuristics) that can identify the scalability problem using measurement data.

5.3.2 Deriving Heuristics

The most critical part of a detection strategy is the heuristics to decide whether a performance & scalability problem is present in the system or not. For each performance problem, symptom, and root cause, we need a detection strategy that accurately identifies the problem. A detection strategy comprises a workload variation, observed metrics, and an analysis strategy all of which contribute to its accuracy as described above.

In order to compare the accuracy of different detection strategies for a certain performance problem, we first define what we understand by accuracy in this context. A detection strategy is a heuristic that based on observed performance data of a system under test, signals if a specific performance problem is present in that system. Based on [3], we define accuracy as a tuple \((1 - r_{fn}, 1 - r_{fp})\), whereby \(r_{fn}\) is the probability that a performance problem is falsely neglected (false negative) and \(r_{fp}\) the probability that a problem is falsely identified (false positive).

A detailed description of deriving and evaluating heuristics for detection of performance and scalability problems is provided in [2]. The rest of this section briefly introduces a number of detection strategies currently implemented in the Dynamic Spotter.

5.4 Software Bottlenecks

During the first year of the project, we focused on a proof of concept of the method on detecting software bottlenecks. The following detection strategies were developed to detect the most typical software bottlenecks.
5.4.1 Directed Growth Strategy

The Direct Growth (DG) detection strategy identifies growing response times over the measurement time of a system under test. It compares response times measured in the beginning to response times measured at the end. If the comparison yields a significant difference, we assume that The Ramp is present. In the following, we describe the experiment setup, the analysis of results and the evaluation of this strategy. We also discuss the weaknesses of this strategy, which ultimately lead to the introduction of the “Time Windows” detection strategy.

To trigger The Ramp, the load driver executes the usage profile defined by software engineers against the SUT. The detection strategy requires only one experiment with predefined duration $D$. During the experiment, the load driver submits a fixed workload intensity $w$ to the SUT, while end-to-end response times are observed. Additionally, for each measured response time an observation time stamp is captured. The result of such an experiment is an ordered series $R = (r_1, ..., r_n)$ of response times with corresponding time stamps $T = (t_1, ..., t_n)$ where $t_i$ is the time stamp of $r_i$ for all $1 \leq i \leq n$.

In order to decide if response times increase over time, the detection strategy divides the response time series $R$ into two subsets $R = (r_1, ..., r_k)$ and $R = (r_{k+1}, ..., r_n)$. The two subsets span approximately equal time intervals so that $t_k - t_1 \approx t_n - t_{k+1}$.

Our evaluations [2] show that for the DG strategy, the error rates are in general high, independent of the workload intensity. Therefore, we investigate an alternative detection strategy for The Ramp based on time windows.

5.4.2 Time Windows Strategy

The Time Windows (TW) detection strategy is based on the observations that i) high workload intensities push The Ramp behaviour faster than low workload intensities and ii) bottleneck effects have to be excluded. The TW strategy addresses both conflicting requirements as described in the following.

To deal with the conflicting requirements, we divide each experiment into two phases: A stimulation phase and an observation phase. During the stimulation phase, the TW strategy pushes a potential The Ramp anti-pattern by submitting a high workload to the SUT. In this phase, no measurements are taken. During the observation phase, the TW strategy applies a closed workload with only one user and a short think time. This workload guarantees that requests are not processed concurrently, allowing us to exclude synchronization problems. In this phase, we capture a fixed number of end-to-end response times.

In order to observe the response time progression during operation, we repeat this experiment increasing the duration of the stimulation phase. In this way, we get $n$ chronologically sorted sets $R_i$ (time windows) each containing a fixed number of response time measurements.

We compare the response times of the SUT for different stimulation times to detect increases in response time during operation. For this purpose, we perform pairwise t-tests on neighbouring time windows. Based on these results of our evaluation [2], we show that we can use the “Time Windows” strategy to detect the anti-pattern The Ramp in a system under test.

5.4.3 One-Lane Bridge

A One Lane Bridge (OLB) [4] occurs, if a passive resource limits the concurrency in an application. Passive resources can be for instance mutexes, connection pools, or database locks. In the following, we introduce the detection strategy for the One Lane Bridge anti-pattern selected due to its low error rate with respect to the reference scenarios.

Since a One Lane Bridge is a typical scalability problem, we are interested in the performance behaviour with respect to an increasing level of concurrency. To detect this anti-pattern, we define a series of experiments observing the end-to-end response time while increasing the number of users for each experiment. The strategy increases the number of users until i) a resource is fully utilized (i.e., its utilisation is larger than 90%), ii) response times increase more than 10 times, or iii) the maximum
number of potential concurrent users is reached. The experiments yield $n$ sets of response times $R_1, ..., R_n$ where $n$ is the number of experiments and $i+1$ is the experiment with the next higher number of users compared to experiment $i$ ($1 \leq i < n$).

In order to distinguish an OLB from a Bottleneck Resource, we additionally measure resource utilization during each experiment. If the SUT contains an OLB, its critical passive resource leads to strongly increasing response times for an increasing number of users. Additionally, CPU utilization is low since the throughput is limited by the passive resource. If the CPU is a Bottleneck Resource (BR), response times increase and CPU utilization is high. Thus, we do not assume an OLB to be present. Finally, if no performance problem occurs, response times and CPU utilization increase only moderately. Strongly increasing response times and low resource utilisation are indicators for an OLB.

5.5 Inter-component Communication Anti-patterns

In the second year of the project, we focused on communication performance anti-patterns. Communication Performance Anti-patterns (CPAs) exhibit poor software performance due to improper messaging, service invocation or method calling behaviour. CPAs become particularly expensive with regards to performance, if single communication steps have to overcome latency in form of computational overhead or network distance. Thus, remote service calls, communication over message oriented middleware (MOM), or database invocations are typical contexts where CPAs occur.

Figure 8 shows an extension of the problem hierarchy with the communication-related performance problems, symptoms and root causes we collected from literature. All occurrences of CPAs come with a high communication overhead which results in high response times under a significant load. In the context of messaging systems the overhead comes from message processing through the messaging clients and the messaging server, such as queuing, serialization, etc. Furthermore, in distributed systems messages have to pass through a network, entailing latency, which directly affects response times. We subsume these overheads under the symptom Excessive Messaging. We consider two different performance anti-patterns which cause excessive messaging: The Blob (God Class) and the Empty Semi Trucks performance anti-patterns. Both anti-patterns result in an unnecessarily high amount and frequency of messages.

Here we provide a brief summary of the new detection strategies that we developed in the second year of the project. We refer the reader to [11] for more details on the patterns, their symptoms, and the corresponding detection strategies.

![Figure 8: Communication Performance Anti-patterns](image-url)

5.5.1 Excessive Messaging

If the messaging intensity in a SUT is independent of the load, the messaging overhead is a constant, which rarely constitutes a performance problem. Let us assume that the messaging intensity directly
depends on the system’s load. In this case, the scaling behaviour of the message throughput with the load is an indicator for excessive messaging. A perfectly scaling system with a minimal messaging overhead shows proportional growth behaviour between the load committed to the SUT and the throughput of messages. In the case of excessive messaging, the message throughput stops growing linearly before the maximal expected load is reached, due to limited passive resources like network bandwidth or message queues. Thus, in order to detect whether messaging in a SUT constitutes a performance problem, we evaluate whether the message throughput grows proportionally with the load committed to the SUT.

5.5.2 Communication Blob
In a distributed system, a Blob anti-pattern from the communication perspective is characterized by a relatively high number of messages transmitted between one single component and all other components. In order to detect a Blob component, the message flow as well as its impact on the performance needs to be analysed. Hence, for each component, we calculate its contributing part to the overall messaging time. If a component constitutes a significant portion of the overall messaging time, we consider it a Blob.

5.5.3 Empty Semi Trucks (EST)
An EST anti-pattern is characterized by a high amount of small messages transmitted between two components as part of a single user request. However, if a component exhibits a high message transmission rate due to a high load, whereas the frequency of message transmissions per user request is low, we do not consider it an EST. Thus, in order to detect an EST anti-pattern we analyse the messaging behaviour of single user requests. For each request, we use thread ids, the method call depth information and the timestamps to reconstruct an annotated call tree instance for each single user request. Information on the message size and the corresponding payload size are attached to each message dispatching method in the call tree. Call tree instances with the same tree structure are grouped and aggregated with respect to the message size information. Therefore, the average message size and payload size are calculated over all call tree instances of a group. Based on this information, we consider messages, which are transmitted in a loop as potential candidates for aggregation. If the message sizes are small, we have a possible indicator for an EST anti-pattern. Based on the aggregated call trees, we calculate for each call of a dispatch method, which has been executed in a loop, the potential amount of saving inefficient bandwidth usage. If the saving potential is higher than the average payload, the EST heuristic reports an occurrence of an EST anti-pattern by pointing to the call path, which contains the guilty method call.

5.5.4 The Stifle
The Stifle anti-pattern is characterized by a significant amount of similar database queries, which are executed as part of a single user request to the SUT. As the characteristics of the Stifle anti-pattern are quite similar to the EST anti-pattern, the Stifle detection heuristic is designed in a similar way as the EST heuristic.

5.6 Summary / Discussion
The Spotting by Measuring method combines systematic search based on a decision tree with goal-oriented experimentation. For this purpose, we are structuring performance and scalability problems known in the literature in a Performance Problem Hierarchy, which guides the search. The method together with its supporting toolset, Dynamic Spotter, allow software engineers to automatically search for performance and scalability problems in an application with relatively low effort. Lowering the burden of performance and scalability validation enables more regular and more sophisticated analyses. The validation can be executed early and on a regular basis, for example, in combination with continuous integration tests. In addition to the evaluation of individual detection strategies, we have applied the Dynamic Spotter to a 3rd party implementation of the well-established TPC-W
D2.2 : Evolution support, second version

benchmark. We have validated the detection strategies and the heuristics using variations of the case study, the results of which are published in [2] and [11]. Using the idea of hierarchically structuring performance problems, Dynamic Spotter was able to stepwise narrow down the root cause of problems from corresponding symptoms. The dynamic, goal-oriented instrumentation approach of Dynamic Spotter kept the measurement overhead low, such that detection results were not impaired by the instrumentation probes. However, our recent evaluation results show that instances of performance problems may hide other performance problems. This issue can be resolved by applying an iterative approach to detect and resolve the problems.

In the following, we discuss the main assumptions and limitations of the approach: The choice of representative usage profiles, the nature of performance problems, PPD’s general applicability (threats to validity), and the usage of heuristics.

5.6.1 Availability of Usage Profiles

In order to execute experiments, the Dynamic Spotter requires a usage profile to generate load on the SUT. A usage profile describes the typical behaviour of the application users. The definition of typical usage profiles (or workloads) is a well-known part of load testing using tools like LoadRunner\(^2\). The effort to define a usage profile can vary depending on the complexity of the application and the required tests. The quality of the usage profile can be critical for the results. While a well-chosen usage profile can trigger rare performance problems, a badly-chosen one may hide even simple problems.

5.6.2 The Nature of the Problems

Although the idea of the Spotting by Measuring method can be applied more generically, the problems we have tested so far with the Dynamic Spotter are tracked to either a specific part in the source code or a specific resource. This may not hold for all such problems. Some are the result of the sheer size and complexity of an application. They are distributed over various places in the source code. In such cases, the Dynamic Spotter may be only able to detect the problem, but not be able to isolate the root causes.

5.6.3 Usage of Heuristics

We acknowledge that the implementation of the method is based on heuristics using best effort. Since a detection strategy can only be falsified by a system or scenario in which it does not correctly identify or diagnose the problem, the detection strategies are only the best with respect to our current knowledge. Furthermore, the definition of new heuristics can require a lot of manual effort and significant expertise in performance engineering. Each detection strategy needs to be evaluated in different scenarios to assess its accuracy and usefulness. However, the combination of goal-driven experiments, heuristics, and systematic search significantly eases the process of performance problem detection and root cause isolation.

\(^2\) [Link to LoadRunner](http://www8.hp.com/us/en/software-solutions/software.html?compURI=1175451#UP0NO3d71lw)
6 The Extractor

In this chapter, we describe how the Extractor is further tested with ENT and SAP use-cases and present the test results.

6.1 ENT use-case

Ericsson Nikola Tesla (ENT) provides the Electronic Health Record (EHR) as a use-case. The EHR is a part of the Ericsson Healthcare Exchange (EHE) platform offered within ENT’s healthcare portfolio. EHE is deployed on a national level in Croatia and several platform services are integrated in many healthcare provider institutions. One of the main components of EHE is EHR, which provides digitally stored health information for patients supporting care, education and research. EHR stores information on medications, past medical history, immunisations, laboratory data, radiology reports, etc. The EHR system, as a part of the EHE platform, needs to share data with healthcare providers, insurance institutions, government agencies, and patients. As such, the EHR system is composed of several services which were used as an input for the Extractor test. Further details about EHR infrastructure, scalability issues, and evolution scenarios can be found in the deliverable D4.2-Requirements and validation [12].

6.2 Testing the Extractor

This section focuses on ENT’s test of the Extractor tool. This is not to be confused with the validation of the tool, which is documented in the deliverable D4.2-Requirements and validation [12]. ENT tested Extractor using the EHR use-case described in the previous section. The Extractor tool was used on modules implementing EHR services, which are organized in 24 Java projects that served as an input to the Extractor tool. The Extractor can be installed using developer or user installation procedure, both of which were tested during the test executed by ENT.

The Extractor was first installed following the user installation procedure. The initial functionality tests included generation of model files from the input source code, and initializing repository and system diagrams from those models. In the initially provided version of the tool there was a missing menu entry that should allow the initialization of the diagrams based on the models generated from the source code. The missing menu entry was added in the following version of the user installation. After the correction of the missing menu error was verified, another issue was noticed during the test of the user installation. The process of generating models ended with the “Workflow failed” error. This behaviour was tested on three machines. The specification of the machines used for the test can be seen in Table 1.

<table>
<thead>
<tr>
<th>Computer</th>
<th>HP EliteBook 8440p_1</th>
<th>HP EliteBook 8440p_2</th>
<th>Samsung NP350V5C-S01</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDD</td>
<td>250 GB</td>
<td>250 GB</td>
<td>750GB</td>
</tr>
<tr>
<td>RAM</td>
<td>8 GB</td>
<td>4 GB</td>
<td>6 GB</td>
</tr>
<tr>
<td>CPU</td>
<td>Intel Core i5, 2.4GHz</td>
<td>Intel Core i5, 2.4GHz</td>
<td>Intel Core i5, 3.1GHz</td>
</tr>
<tr>
<td>OS</td>
<td>Win 7</td>
<td>Win 7</td>
<td>Win 7</td>
</tr>
<tr>
<td>Java version</td>
<td>1.6.0.25, updated to 1.7.0.55</td>
<td>1.7.0.51</td>
<td>1.7.0.55</td>
</tr>
</tbody>
</table>

Also, other than trying the process on several machines, a test was performed using different Java projects, in order to assure the error is not a result of the specific project source code issue. The problem was reported, and at the stage of the test during which the problem still wasn’t resolved, it was decided to test the developer installation in parallel. The summary of the developer installation test results regarding the 24 Java projects that served as an input can be seen in the Table 2.
Table 2: The summary of the Extractor developer installation test results

<table>
<thead>
<tr>
<th>Test outcome</th>
<th>Number of Java projects used as a test input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workflow failed</td>
<td>11</td>
</tr>
<tr>
<td>OK</td>
<td>11</td>
</tr>
<tr>
<td>NullPointerException</td>
<td>2</td>
</tr>
</tbody>
</table>

The Workflow failed error appeared again in case of developer installation (using 11 of 24 projects as an input to the Extractor tool), although it was caused by different exception (WorkflowFailedException caused by ClassCastException) than the one seen in error user installation (ArrayStoreException). The Null Pointer Exception was found to be the result of tests in which the input project was not correctly recognized as a Java project by the Eclipse IDE, and as such it was not further processed as a tool error. The rest of the projects resulted with generated models from which the diagrams were initialized, which was the expected outcome of the test.

After the initial functionality tests, additional tests were executed focusing on the composition and merging of the components detected from the source code. The developer installation of the tool was used for this test. Extractor offers the possibility to change the weights of the parameters that affect how the tool identifies an individual component from the source code. The parameters observed in the test were Clustering Merge Threshold Min, Clustering Merge Threshold Max, Clustering Composition Threshold Min, and Clustering Composition Threshold Max. Components composition and merging were tested on the source code of two EHR modules. Two test cases were executed for each module, varying the number of components identified from the source code. In the first test case used on the Appointment Scheduling project, using the parameter values from the Table 3, test resulted in 35 identified components, and 34 interfaces. In the second test using the same input project, parameters were changed to increase the merging of the components, resulting with one component (compared to 35 in the previous test), but the number of interfaces remained the same. From 34 generated interfaces, only three of them were connected to the component, leading to conclusion that Extractor adjusts the generation of components according to the merge and composition configuration parameters, but the interfaces are not modified accordingly. It is presumed the interfaces should also be composed or merged, and connected to the composed component. The second project used for test was EMH module. Using this project as an input, both test cases resulted with interfaces connected to the component, i.e., there were no generated interfaces that weren’t linked to any of the components, although it was noticed that some of the requiring interface connections were lost when there was only one component generated. The parameter weights used in the described tests and the test results can be seen in Table 3.

Table 3: Parameters and results of the composition/merging tests

<table>
<thead>
<tr>
<th>Test Case (Project Name)</th>
<th>Parameters weights</th>
<th>Generated components</th>
<th>Generated interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appointment Scheduling</td>
<td>Clustering Merge Threshold Min = 100 Clustering Merge Threshold Max = 100 Clustering Composition Threshold Min = 0 Clustering Composition Threshold Max = 100</td>
<td>35</td>
<td>34, all of the interfaces are connected to at least one of the components</td>
</tr>
<tr>
<td></td>
<td>Clustering Merge Threshold Min = 45 Clustering Merge Threshold Max = 100 Clustering Composition Threshold Min = 25 Clustering Composition Threshold Max = 100</td>
<td>1</td>
<td>34, three interfaces are connected to the generated component, rest of them are not linked to any component at all</td>
</tr>
</tbody>
</table>
The details on ENT’s Extractor tool validation results can be found in the deliverable D4.2 Requirements and validation.
7 Conclusions and future work

Overall, WP2 achieved the setup goals for the reporting period to a large extent. All tools are available, have been tested and results have been gathered. Furthermore, the tools are integrated into the CloudScale Environment, which makes them available to arbitrary end-users. The Static Spotter has fewer search patterns modelled than originally planned at this stage, but we had to spend more effort in modernising the tools platform than initially expected.

In sum the delays from Y1 have not been fully compensated despite the mitigation steps taken. In addition, the move to Chemnitz caused further delays in the final month of the second period and the first month of the third; these had particular impact on this deliverable.

In Y3 we plan to further stabilise and extend the tools, e.g. with support for more patterns and anti-patterns.
References


[12] Deliverable 4.2, Requirements and validation, second version