Scalability management for Cloud Computing

Seventh Framework Programme:
Call FP7-ICT-2011-8
Priority 1.2Cloud Computing
Collaboration Project (STREP)

Deliverable ID:	Preparation date:
D2.1	31 October 2013

Milestone: Released

Title:

Evolution support, initial version

Editor/Lead beneficiary (name/partner):
Jinying Yu / UPB

Internally reviewed by (name/partner):
Darko Huljenic/ENT

Approved by:
Executive board

Abstract:

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.
- Modelling and analysing scalability anti-patterns (HowNotTo, bad practices).
- Extractor methodology.
- Scalability analysis by systematic experimentation.
- Use cases of scalability anti-pattern detection.

Dissemination level		
PU	Public	x
CO	Confidential, only for members of the consortium (including Commission Services)	

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.

CloudScaleconsortium

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 zaBrdom 100
1000 Ljubljana, Slovenia
www.xlab.si

Contact: Jure Polutnik
jure.polutnik@xlab.si

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

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

Table of contents

List of figures

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 include:

- Overview of the evolution support. In this chapter, we describe how we support software evolution within the CloudScale method and its processes. There are four tools in the processes. The Extractor extracts software architecture from source codes and visualizes with ScaleDL models. The Analyser is a tool that can automatically check whether a modelled system meets the scalability requirements in the requirement specification. The Dynamic Spotter finds components responsible for hindering scalability by inspecting source codes, while the Static Spotter inspecting software architecture.
- Modelling and analysing scalability anti-patterns (HowNotTo, bad practices). In this chapter, we present our idea of how to model and analyze scalability anti-patterns as well as provide a concrete example, “the Blob” anti-pattern, and model it in our anti-pattern template.
- Extractor methodology. In this chapter, we talk about our approach of the Extractor and the generated result from the Extractor, the partial ScaleDL model.
- Scalability analysis by systematic experimentation. In this chapter, we introduce our idea of the Spotting by Measuring method and its complementing tool, the Dynamic Spotter.
- Use cases of scalability anti-pattern detection. In this chapter, we briefly present a summary of our achievements with validation of the Spotter and our plan to apply in the context of the SAP use case.

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).

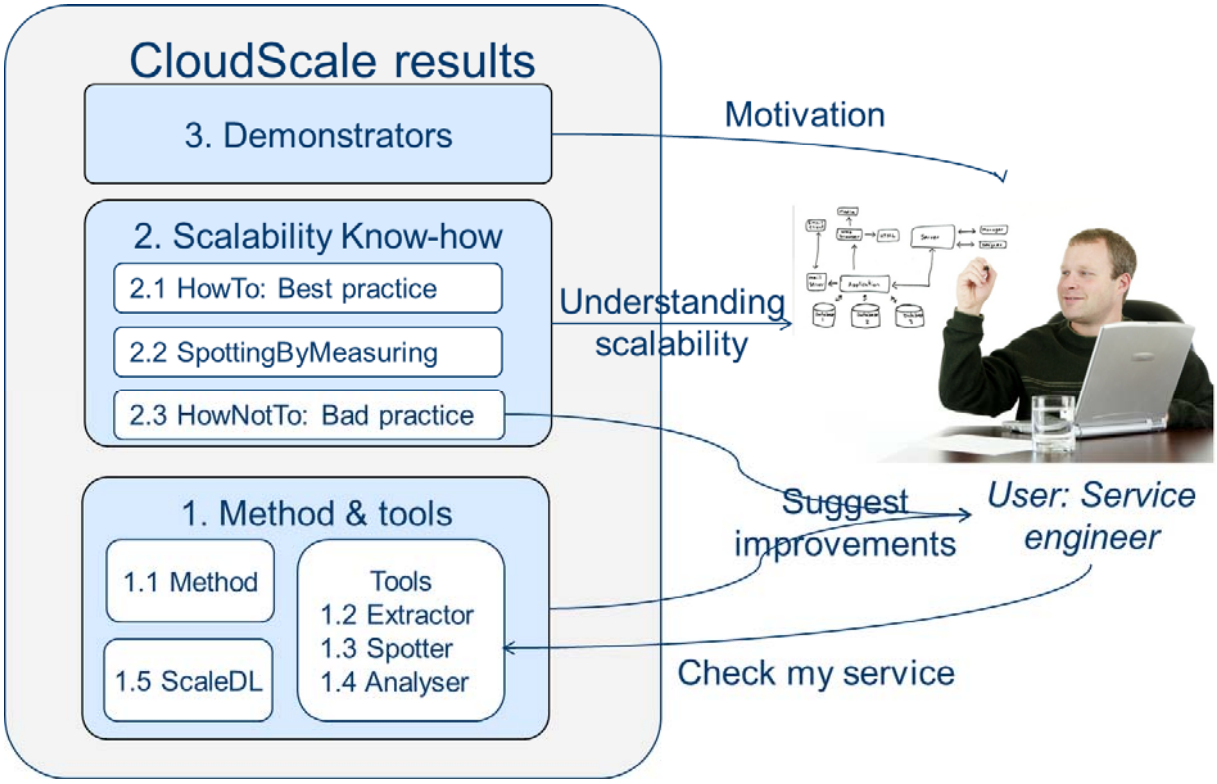


Figure 1.1: The results of CloudScale

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 Extractor presented in this document relates on the following deliverables:

- D1.1 – Design Support: this document presents the overview of the CloudScale method.
- D3.1 – Integrated tools: this document presents integration of the Extractor into the CloudScale tool set.
- D5.1 – First version of Showcase: Presents the CloudScale showcase which builds on the TPC-W benchmark. This benchmark is also used as an example in this deliverable.

1.3 Contributors

The following partners have contributed to this deliverable:

- University of Paderborn
- SAP

1.4 Acronyms and abbreviations

PCM	Palladio Component Model	PoC	Proof of Concept
PINOT	Pattern INference recOvery Tool	SoMoX	SOftware MOdel eXtractor

2 Overview of evolution support

In this chapter, we describe the overview of the evolution support in the CloudScale method and the process as well. This chapter is contributed by UPB.

2.1 Evolution support in the CloudScale method

In this section, we present an overview of the evolution support in the CloudScale method.

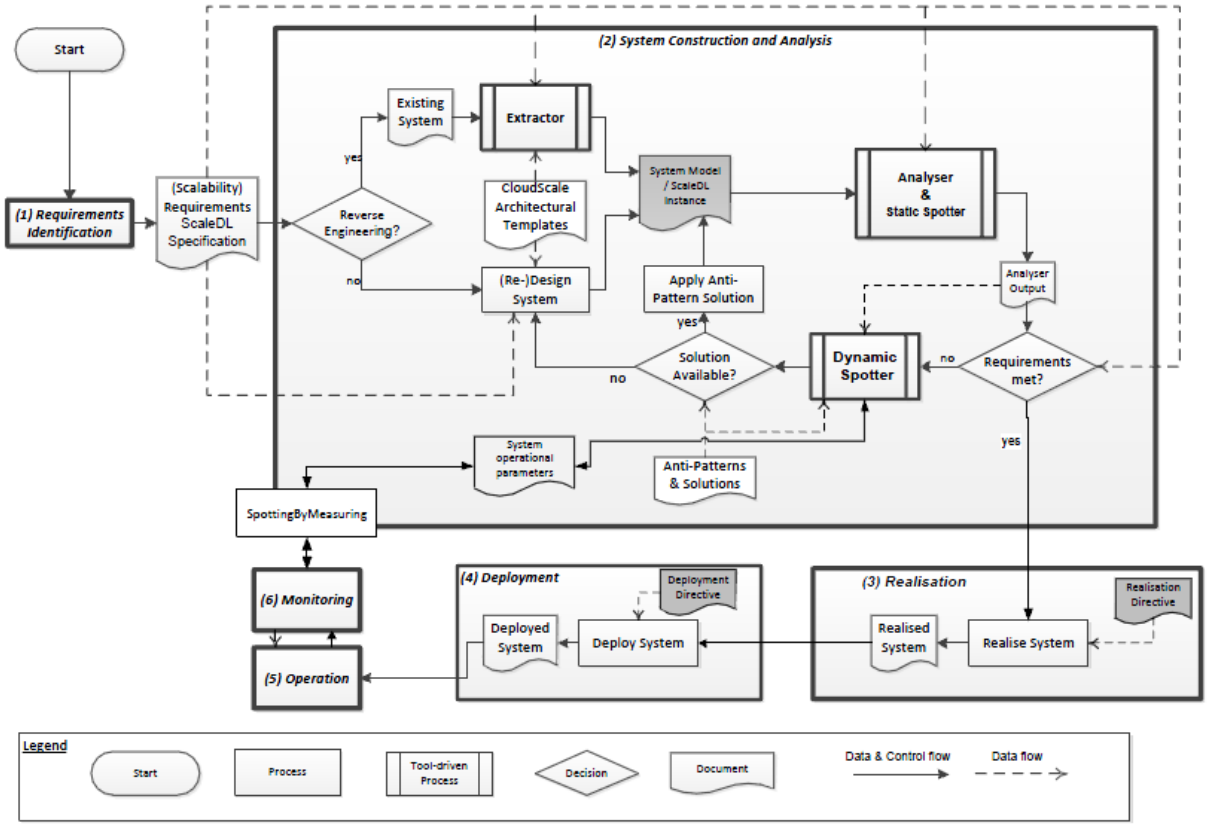


Figure 2.1: CloudScale method overview

Figure 2.1 is the overview of the CloudScale method. The part of system construction and analysis shows the evolution support of the CloudScale method. There are four tools involved in the evolution support, the Extractor, the Analyser, the Dynamic and Static Spotter. The Extractor is a tool that extracts software architecture from source codes and visualizes with PCM models. The Analyser is a tool that can automatically check whether a modelled system meets the scalability requirements in the requirement specification. (See more details of the Analyser are in Deliverable 1.1) The Dynamic Spotter finds components responsible for hindering scalability by inspecting source codes (See more details in section 5), while the Static Spotter inspecting software architecture. The Static Spotter is not yet developed. We will work on the Static Spotter from its concept and a prototype in Year 2.

2.2 Processes

In this section, we describe the whole processes of evolution support. Assume we have an outdated online book store system. Unfortunately, there are no documents about the book store system, no UML diagrams, no specifications, and no architecture drafts. The only artefact available is the source code. Now we would like to eliminate potential scalability issues from the system before we deploy it into the cloud computing environment.

As the first step, we use the Extractor to work on the existing book store system's code. The Extractor parses the source codes and generates PCM models from the source code. Now we have the static software architecture of the book store in PCM models.

Based on the PCM models, we have two tools to cope with two scenarios.

One scenario is that we have new requirements coming in. Let's imagine originally the book store system is designed to serve 1000 customers with a response time of the most 2s. Now we want to know if the system could still offer a response time within 2s while serving 2000 customers at the same time. We use the Analyser to check whether the PCM model meet the new requirements or not. If not, we use the Static Spotter to find the possible Anti-patterns. When the Static Spotter finds anti-patterns, it could propose possible anti-patterns solutions and apply the selected solution. When the Static Spotter can't find any anti-patterns, we let the Dynamic Spotter spot anti-patterns by measurement. If the Dynamic Spotter can't have a solution either, we have to consider redesigning the system in order to fulfil the new requirements.

In the second scenario, even when there is no new requirement coming in, we can use the Static Spotter to work directly on the generated PCM model or use the Dynamic Spotter to inspect the source code to proactively check whether or not there are anti-patterns existing. If there are, the Static and Dynamic Spotters then propose possible solutions and apply the selected solution on the models.

In either scenario, we could eliminate the scalability issues from the book store system through our evolution support processes.

The Static Spotter is not yet developed. We plan to finalize the concept and a prototype in Year 2. We describe more details of anti-patterns in Section 3 and the Dynamic Spotter in Section 5.

3 Modelling and analysing scalability anti-patterns (HowNotTo, bad practices)

This chapter presents how we model and analyse scalability anti-patterns. We describe one anti-pattern in a formulate way and our strategy to detect anti-patterns in the following sections. This chapter is contributed by UPB.

3.1 Scalability anti-patterns

In this section we discuss scalability anti-patterns. Patterns originate from a work of an architect named Christopher Alexander. He developed a pattern language and used the pattern language for town planning and building designing [Alexander]. A pattern language is a language which defines a collection of patterns. A pattern is a common solution to a recurring problem. In software development, we use the same concept of patterns so that we do not need to reinvent the wheel. Patterns in software development are known as best practices, proven solutions to common problems. Software engineers categorize patterns and put them into catalogues so that they could be looked up and reused.

Anti-patterns are conceptually similar to patterns. They both document common solutions to recurring problems. While patterns provide proven solutions with positive effects, anti-patterns record those solutions with negative consequences. Anti-patterns show software engineers what to avoid as well as solutions. Therefore, anti-patterns are also known as bad practices. Software engineers find anti-patterns useful because with anti-patterns they could identify common mistakes and correct or avoid them. This is no exception for performance anti-patterns, "because good performance is absence of problems." [C. Smith] Performance anti-patterns describe "bad practices" which hinder responsiveness and scalability. Because hindered responsiveness has potential negative effect on scalability especially when software is scaling up, we consider performance anti-patterns as well as scalability anti-patterns.

In the following section we try to formalize performance and scalability anti-patterns using the template below:

Name: the anti-pattern's name

Also known as: the anti-pattern's other known names

Example: an example shows the anti-pattern

Problem: what is the recurrent situation that causes negative consequences? [C. Smith]

Solution: How can we avoid, minimize or refractor the anti-pattern? [C.Smith]

Detection: How can we detect the anti-pattern?

Variants: Are there known variants of the anti-pattern?

Consequences: What are the negative consequences?

See also: literature references

3.2 Anti-pattern Catalogue

We intend to have a catalogue of scalability anti-patterns. In Year 1, we demonstrate one anti-pattern by using our template. We'll describe more anti-patterns in the coming year.

3.2.1 Anti-pattern "the Blob"

Name: the Blob

Also known as: the god class

Example: assume we have a process controller program. There are Controller and Valve Class as Figure 3.1 shows. This example is from [C. Smith].

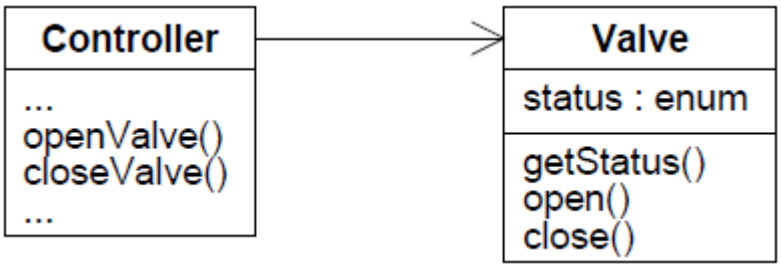


Figure 3.1: Excerpt from a class diagram for a process controller program [C. Smith]

Problem: this problem occurs when one class performs most of the system work relegating other classes to minor, supporting roles.

As in the example, the Controller does all the system work, openValve and closeValve. The Valve simply reports its status (open or closed) and answers the Controller's invocation, open and close respectively. The Controller requests information, make decisions and orders the Valve to open or close.

For example, if the Controller wants to open the valve, it must get the current status from the valve first and tell the valve to open. See the code below:

```
void openValve() {
        status currentStatus;
        currentStatus = theValve->getStatus();
        if (currentStatus != open)
        theValve->open();
}
```

In the case when the valve is closed, it costs two messages to open the valve including one unnecessary message. This could potentially lead to unnecessary message traffic, which hinders scalability.

Solution: keep related data and behaviour together. An object should have most of necessary data to make a decision.

As in the example of the Controller and the Valve, one possible solution to reduce the messages is to check the valve status locally inside the Valve class (keep related data and behaviour together).

So the openValve() operation in Controller is changed to:

```
void openValve() {
        theValve->open();
}
```

and the open() operation in Valve becomes:

```
void open() {
        if (status != open)
        status = open;
}
```

Now it only requires one message to open the valve.

Detection: unknown

Variants: unknown

Consequences: the anti-pattern potentially causes excessive message traffic.

See also: [C. Smith]

We find this problem in TPC-W that is an online book store. When we want to add a new customer, the servlet calls Database class to check if the customer is in the database first; if not, it calls Database to add the new customer. See the pseudo code in the servlet class below:

```
void addNewCustomer(){
        Boolean isNewCustomer;
        isNewCustomer = Database->isCustomerinDatabase();
        if (isNewCustomer == TRUE)
        Database->addCustomer();
}
```

3.3 Anti-pattern detection strategy

We plan to detect the scalability anti-patterns in two ways with two tools, the Static Spotter and Dynamic Spotter.

One is we use the Static Spotter to work on the partial ScaleDL model. We extract the partial ScaleDL model directly from the source code, then the Static Spotter analyses the model. By using the formalised anti-patterns, we try to locate the anti-patterns in the partial ScaleDL model and apply the solutions accordingly. There are existing detection strategies we could consider. Nija et al.[5] describe PINOT, a design pattern detection tool that could find structure- and behaviour-driven design patterns by parsing the Java Abstract Syntax Tree (Java AST) file. In CloudScale, the Extractor also generates Java AST files. Another possible solution is to use the rule-based detection strategy to find anti-patterns. Stoianov et al.[6] use Prolog predicates to describe anti-patterns. They compile the rules into a fact database to find all possible combination of facts while matching. In CloudScale, Archimetrix that the Extractor is based on has a rule-based engine. While the engine is originally designed to find design deficiencies, we could extend it to detect anti-patterns as well.

The other is the Dynamic Spotter detects anti-patterns by measuring. The Dynamic Spotter spots scalability issues by measurements and recognize anti-patterns. We explain this approach in more details in Section 5 We explain this approach in more details in Section 5.

4 Extractor methodology

In this chapter, we describe the methodology of the Extractor and present the tool generated result. This chapter is contributed by UPB.

4.1 Methodology description

Our tool, the Extractor, is based on the Archimetrix approach [7]. Archimetrix is a tool-supported reengineering process that combines different reverse engineering approaches to enable an iterative recovery and reengineering of component-based software architectures. Figure 4.1 shows the oview of the Archimetrix process. Archimetrix combines clustering-based architecture reconstruction and pattern detection techniques to recover the architecture of a software system from source code. While the clustering extracts a software architecture based on source code metrics (Step 1 and 2), the pattern detection is used to detect design deficiencies in the architecture which influence the architecture reconstruction negatively. Archimetrix supports the reengineer by identifying components that are especially relevant for the detection of design deficiencies (Step 3 and Step 4). It also ranks the detected deficiencies so the reengineer can focus on the important problems (Step 5). Once the deficiencies are removed (Step 6), the clustering can be used again to get a clearer view of the improved software architecture. The Extractor is focused on parsing and architecture reconstruction (Step 1 and 2).

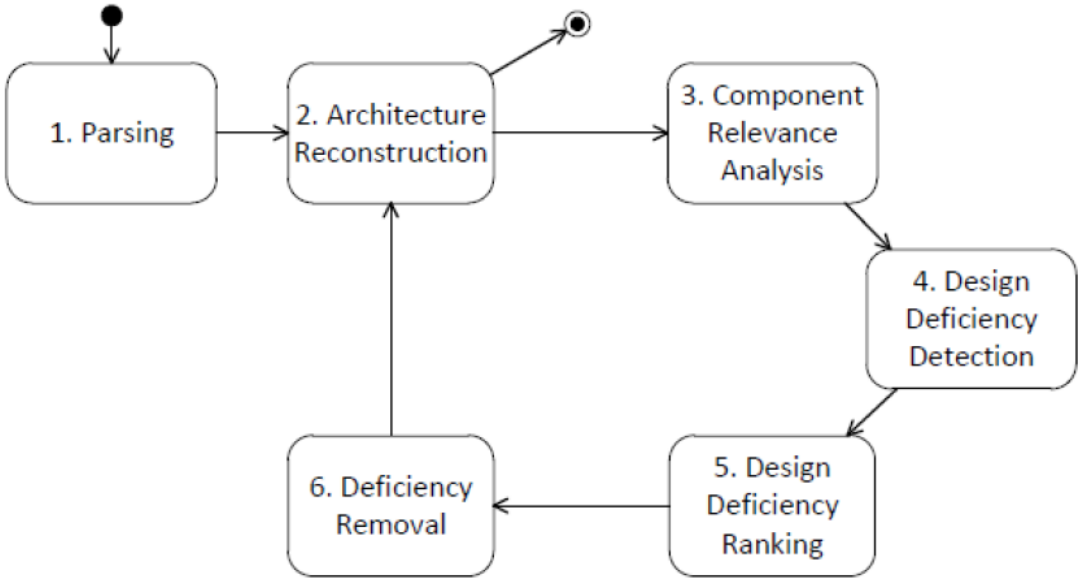


Figure 4.1: Overview of the Archimetrix process [9]

4.2 Integrate architectural model with scalability measurement

Generally there are two different approaches to evaluate software performance, model-based and measurement-based approaches. Model-based approaches firstly build up performance or scalability models and then use analytical methods or simulation. One of the biggest advantages of model-based approaches is that it is applicable on early software development cycles in which typically software under development does not run at all. In model-based approaches, the accuracy of performance models is critical. Inaccurate models will lead to inaccurate analyses. On the other hand, measurement-based approaches are often more accurate because these approaches monitor real systems' implementation on the expenses of monitoring overhead.

In the CloudScale, we try to integrate architectural models, the Palladio Component Model, with scalability measurements. In this way, we could not only support scalability analysis at the early development stage but also monitor real systems' implementation.

"The Palladio Component Model captures the software architecture with respect to static structure, behaviour, deployment/allocation, resource environment/execution environment, and usage profile. In the Palladio Component Model, software is described in terms of components, connectors, interfaces, individual service behaviour models (so-called Service Effect Specifications, SEFF), servers, middleware, virtual machines, network, the allocation of components and servers, models of the user interaction with the system, etc. Overall, the PCM captures multiple views of software systems including elements which affect the extra-functional properties (e.g. performance, reliability) of software systems." [8]

4.3 Example: Extracting ScaleDL model of the showcase TPC-W

In this section, we demonstrate how the Extractor generates the partial ScaleDL model. We use the Extractor to generate the partial ScaleDL model from the source code of the TPC-W. The TPC-W is an online book store web application. It is developed in Java. The Extractor works in two steps. Step 1 parsing: the Extractor uses the MoDisco framework to parse the source code and creates .xmi files as shown in Figure 4.2.

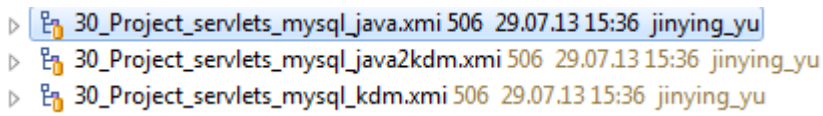


Figure 4.2: Step 1 Parsing

Step 2 architecture reconstruction: we need to configure the SoMoX[10] settings (as shown in Figure 4.3) before the Extractor generates the partial ScaleDL model from .xmi files. The SoMoX settings are meant to define the rules such that how the Extractor composes and merges components (clustering composition threshold and clustering merge threshold).

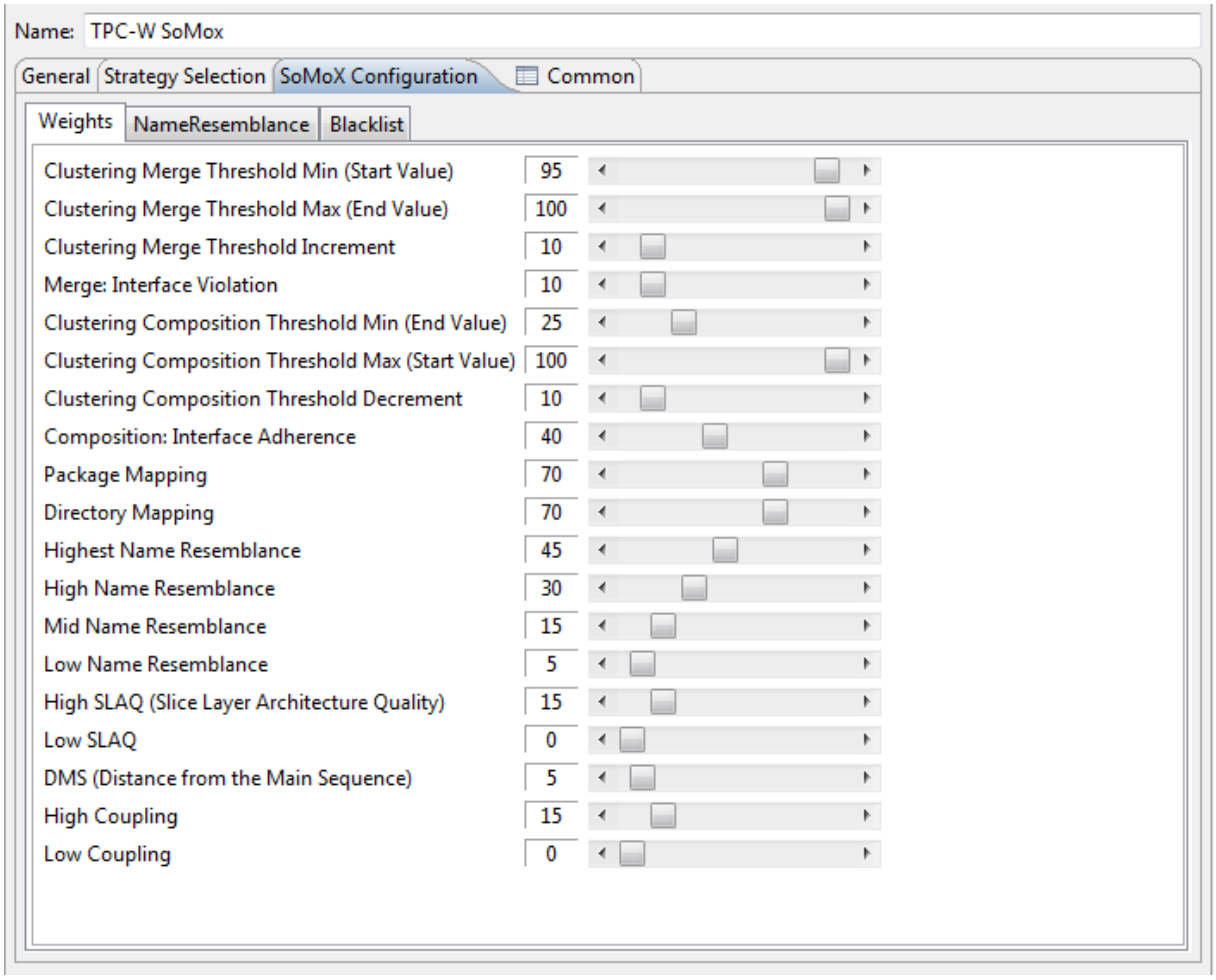


Figure 4.3: Step 2 configure SoMoX setting

After configuring all the settings, we run the Extractor to generate the partial ScaleDL models. The results are shown in Figure 4.4.

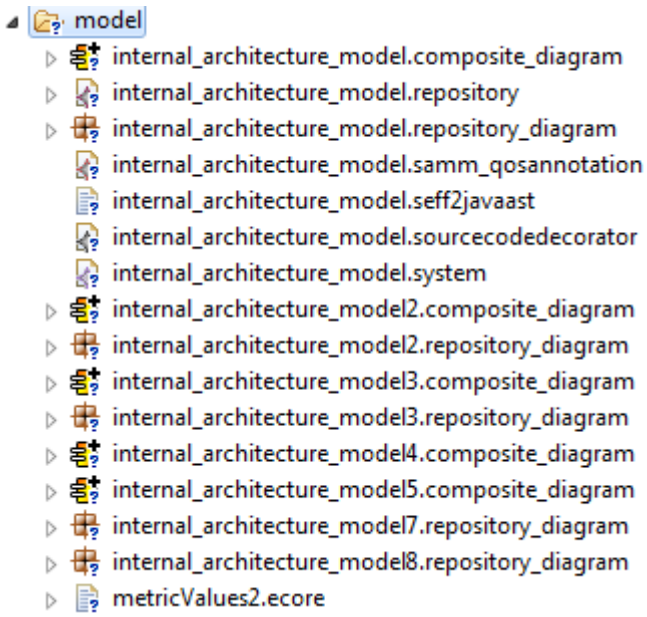


Figure 4.4: Model files generated by the Extractor

As we open the repository diagram, we can find there are 18 primitive components and 4 composite components. In Figure 4.5, it shows 4 primitive components which are from servlet classes.

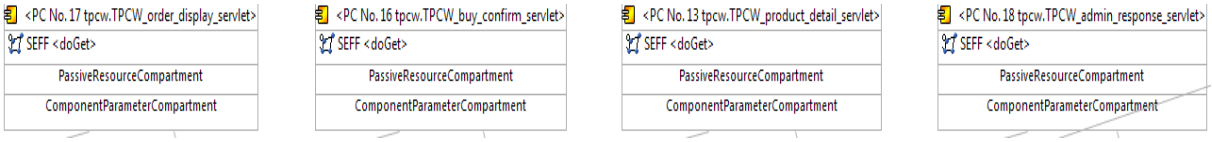


Figure 4.5: Excerpt 1 from the TPC-W repository diagram

In Figure 4.6, it shows the Database component provides a list of database operations. Whenever other components want to access the database, they call the provided operations.

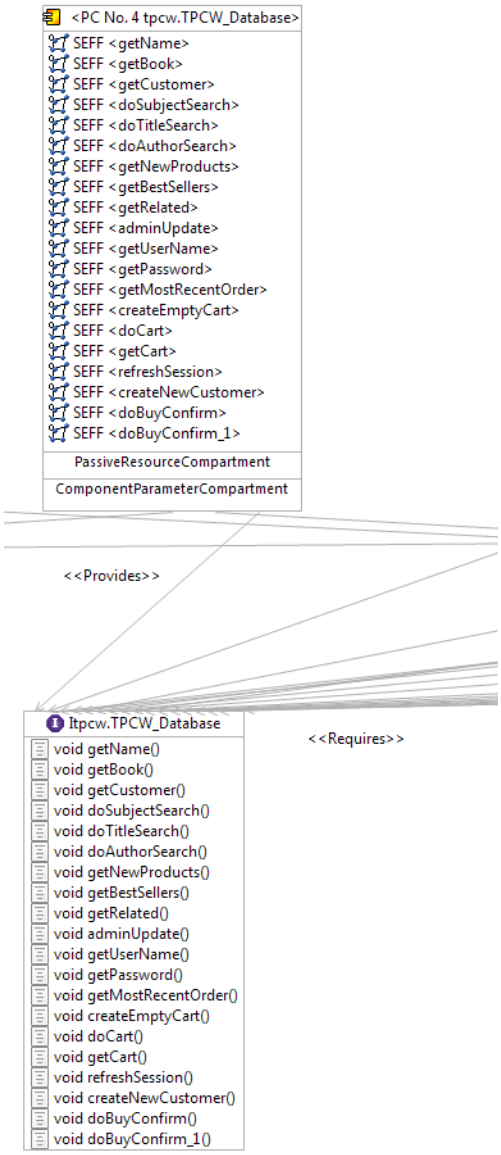


Figure 4.6: Excerpt 2 from the TPC-W repository diagram

4.4 Integration into the CloudScale tool set

The tool Archimetrix will integrate into our CloudScale tool set. The further details are given in Section 3 of Deliverable 3.1.

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. This chapter is contributed by SAP.

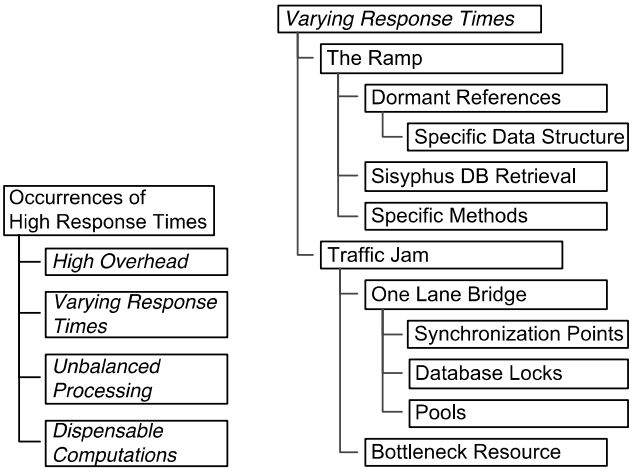
5.1 Overview

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 Performance 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.



(a) Symptoms of known performance problems. (b) Performance problems causing *Varying Response Times*.

Figure 5.1: Excerpt of the Dynamic Spotter Performance Problem Hierarchy

The hierarchy is structured in categories, symptoms, performance problems, and root causes. The category Occurrences of High Response Times in Figure 5.1(a) 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 5.1(b) shows the performance problem hierarchy for Varying Response Times. We identified the performance 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 a 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 performance 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 problems, symptoms, and root causes.

5.3 Detection Strategies

To detect performance 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 performance problems), our algorithm looks for more and more specific performance problems 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 & scalability problem and

root cause, we require a detection strategy. In this section we describe what detection strategies are and how they are used.
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 performance 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 performance problems exist in the SUT as a whole, the latter identifies the root cause of a performance 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 performance problem. We define the required rules for dynamic instrumentation and workload variation as well as analysis methods (heuristics) that can identify the performance problem using measurement data.

5.3.1 Deriving Heuristics for Performance Problem Detection

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.

5.3.2 Directed Growth (DG)

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.

5.3.3 Time Windows (TW)

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.

5.3.4 Detection Strategy for "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.

5.4 Summary / Discussion

The Dynamic Spotter 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 Dynamic Spotter allows 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 are applying the Dynamic Spotter to a 3rd party implementation of the well established TPC-W benchmark. So far, by fixing the detected problems, we were able to increase the maximum throughput of the benchmark from 1800 requests per second to more than 3500. As such, the performance problems had a significant effect on the benchmark results. In the future, based on the encouraging results achieved so far [2], we plan to integrate our approach with the development infrastructure at SAP. This will allow us to stepwise improve and refine our approach and extend the range of performance problems that can be identified.

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.

5.4.1 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[1]. 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.

5.4.2 The Nature of Performance Problems

The performance and scalability problems we have tested so far with the Dynamic Spotter can be tracked to a specific part in the source code or a specific resource. However, 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 cannot isolate the root causes.

5.4.3 Usage of Heuristics

We acknowledge that the Dynamic Spotter 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,

[1] http://www8.hp.com/us/en/software-solutions/software.html?compURI=1175451#.UP0NO3d71Iw

heuristics, and systematic search significantly eases the process of performance problem detection and root cause isolation.

6 Use Case of Scalability Anti-patterns Detection

In the context of CloudScale, scalability anti-pattern detection is supported by three tools: the Dynamic Spotter, the Static Spotter, and the Analyzer. During the first year of the project, we developed a PoC of the Dynamic Spotter. In this section, we briefly present a summary of our achievements with validation of the Dynamic Spotter and our plan to apply in the context of the SAP use case. This chapter is contributed by SAP.

6.1 TPC-W

The first versions of the Dynamic Spotter will be applied to a 3rd party implementation of the well-known TPC-W benchmark. So far we have deployed the benchmark in two environments. The Dynamic Spotter, as its current version, can identify four performance problems that are located in the benchmark implementation, the web server, the database, and the infrastructure. By fixing these problems, we have been able to increase the maximum throughput of the benchmark from 1800 requests per second to more than 3500. We will continue with validation of future versions of the Dynamic Spotter on the modified versions of the TPC-W deployed on a Cloud platform. (see Deliverable D5.1, Section 5.2.2)

6.2 SAP Cloud Applications

The main use case of the Dynamic Spotter will be in the context of building scalable Cloud services and applications within SAP, with a focus on *data-centric applications*. A data centric application is one in which the database plays a key role, the application code is more generic and (almost) all business logic is defined through database relations, constraints, and queries. Built on top of the SAP HANA Cloud platform, the use case will demonstrate how the Dynamic Spotter can identify scalability bottlenecks of SAP HANA Cloud applications. We are currently experimenting with the setup of the required infrastructure to apply the Dynamic Spotter on SAP's internal Cloud applications. During this process, we will identify a suitable internal application which will serve as the main use case of applying the Dynamic Spotter within SAP's Cloud infrastructure.

7 Conclusion/Further work

In this deliverable, we present our initial work on the evolution support within the CloudScale.

The Static Spotter identifies scalability anti-patterns from the model generated by the Extractor and the Dynamic Spotter finds the anti-patterns by measuring. The Extractor R1.2 is modernized and the initial version is created and tested with TPC-W. We plan to further test it with SAP and ENT use cases. The Dynamic Spotter R1.3 is created and tested with TPC-W. The result shows the tool could identify scalability issues. This also means our method SpottingByMeasuring R2.2 is capable to spot scalability issues by measurements because the Dynamic Spotter is based on the method SpottingByMeasuring. While the Static Spotter is not yet developed, we will finalise the concept and create a prototype in Year 2. As for HowNotTo (Bad practice) R2.3, we have defined a template for scalability anti-patterns and we demonstrate the template by formalising one anti-pattern “the Blob” with the template.

References

[1] CloudScale EU FP7 project, www.cloudscale-project.eu/, last accessed 2012-12-04

[2] A. Wert, J. Happe, and L. Happe, “Supporting Swift Reaction: Automatically Uncovering Performance Problems by Systematic Experiments”, in Proceedings or the International Conference on Software Engineering (ICSE), 2013.

[3] J. Swets, “Measuring the accuracy of diagnostic systems,” Science, vol. 240, no. 4857, pp. 1285–1293, 1988.

[4] C. Smith and L. Williams, “Software performance anti-patterns,” in *Proc. WOSP*. ACM, 2000, pp. 127–136.

[5] Nija Shi and Ronald A. Olsson. Reverse engineering of design patterns from java source code. In ASE ’06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, pages 123–134, Washington, DC, USA, 2006. IEEE Computer Society.

[6] Alecsandar Stoianov and Ioana Sora. Detecting patterns and antipatterns in software using prolog rules. In Computational Cybernetics and Technical Informatics (ICCC-CONTI), 2010 International Joint Conference on, pages 253–258, Department of Computers, Politehnica University of Timisoara, Romania, 2010.

[7] M. Platenius, M. von Detten, S. Becker: Archimetrix: Improved Software Architecture Recovery in the Presence of Design Deficiencies. In Proceedings of the 16th European Conference on Software Maintenance and Reengineering, pp. 255 - 264. IEEE, March 2012

[8] http://www.palladio-simulator.com/science/palladio_component_model/

[9] Markus von Detten. Reengineering of Component-Based Software Systems in the Presence of Design Deciencies Dissertation Software Engineering Group, University of Paderborn, January 2013.

[10] http://sdqweb.ipd.kit.edu/wiki/SoMoX