Abstract

This white paper describes the CloudScale Method, a method for designing scalable, elastic and efficient services. We also outlines the granularity trade-off between level of detail and how this affect prediction precision, accuracy and scope on the one hand versus and manuell and computerized effort on the other hand.

1 Objectives

This section describes objectives at three levels: for this white paper, for using the CloudScale Method and more detailed about granularity when executing the CloudScale Method.

1.1 Objectives of this White Paper

This white paper can be read by at least two different readers:

- System engineers with no prior performance engineering background. They will have difficulties with standard performance engineering manual steps, like completing a PCM for resource environment, allocation model, and usage scenario. Eclipse has very many possibilities, but for CloudScale only a few are valid. A walkthrough with a simple example demonstrating the method and how it interact with the tools, would help. This objective is achieved by the CloudScale user guide.

- Experienced system engineers with performance engineering background: Illuminating the trade-off between accuracy and effort would help all CloudScale users, also the more experienced. This is the main focus of this white paper.

An important part of this method is the manual steps like setting objectives for scope and accuracy, setting configuration parameters, instrument source code or finding out you are satisfied or not. Which guidance is available for performing them and how do you know if you have to adjust something, and then what shall you adjust? We will try to answer these questions using the concept of granularity as a common theme in this white paper.

1.2 Overall Objectives

Before starting the CloudScale Method we must have a more or less clear idea of what we shall find out. Possible objectives are:

- Trade-off between cost, functionality and requirements (quality) during development. We may for example compare two different architectures.

- Trade-off between cost, functionality and requirements (quality) during modification. The new parts can be a new operation, or a new architecture. A new architecture may simple mean to compose existing services and components differently, but it may also mean replacing some of them.

- Compare scalability, elasticity and efficiency of competing services.

- Compare competing deployments for an existing service.

Common for all these objectives is that we need to do some sort of scalability, elasticity or efficiency analysis. The CloudScale Method is a method for performing such analysis. When we do this analysis have a granularity trade-off. Generally, answering a more detailed question, like finding an optimal deployment requires more details, compared to finding the best of two competing architectures. We will now look at granularity in more detail.

1.3 Objectives on Granularity

By granularity we mean level of detail. A coarse grained model has few details compared to a fine-
grained model, for example. We will describe this in more details for each method step, but more generally the level of detail will have consequences both for what we put into the analysis as well as what we get out of the analysis.

For what we put into the analysis, two aspects related to granularity, or level of detail, are important:

- Amount of manual work, or effort used to follow the method. To make a scalability model of a service consists of several manual steps, and generally the effort is related to the sophistication of the model. The level of detail of a model increase when you model more operations, more components, more resources and more complex relation between them.

- Runtime to do the analysis on the computer. Most often this time will be negligible, but for some analyses we may spend several hours and then this time will also influence on the amount of manual work. The amount of instrumentation will often be related to run time, because the more instrumentation you have, the higher the run time will be.

When it comes to what we get out of the analysis three aspects of the result are important and are also related to granularity, or level of detail:

- Precision, relating to the repeatability of the results, normally quantified by a confidence interval [9]. To increase precision (i.e. reducing confidence intervals), we can run a simulation several times (with different seeds).

- Accuracy, the difference between the reported value and the "real" value [9]. One way of increasing accuracy is to measure the same value several time, so that statistical variation is accurately modelled. This will increase the accuracy, while reducing the precision. Validation where we compare the "actual" service with the modelled service is the key to increase accuracy.

- Scope, relating to coverage. We may look for scalability problems in the complete service as well as all its underlying services, a broad scope, but we may also confine our scalability investigation to one of the many classes inside of the service, a narrow scope. Similarly, we may focus on the use of processing resources, and ignore the use of storage resources. We may also focus on one critical operation and ignore the remaining operations.

Generally, the more you put into the method in terms of effort and run time, the more detailed your model will be and the more you get out in terms of precision, accuracy and scope. A detailed approach gives good precision but with a high effort, whereas a coarse approach gives low precision with a low effort. An important part of the method is to shed light on this granularity trade off. The level of granularity should be sufficient for meeting this objective, but not more detailed, because then it will also be too costly in terms of manual effort.

This granularity trade-off is important, because the manual effort involved using our tools as well as our methods is the showstopper for their widespread use. Using more coarse-grained models, this manual work may be reduced, but then at the cost of precision, accuracy and scope. The question then becomes: which precision, accuracy and scope are required for spotting scalability/elasticity/efficiency problems in a software service? The answer may not be simple. For example, more accurate instrumentation/models may be required to spot elasticity problems than what is required to spot scalability problems. Probably a baseline scalability model will be required first also with validation, before it is meaningful to analyze elasticity and efficiency. The more concrete advice we can provide, the better.

We will recommend a coarse approach first and afterwards increase the granularity of key parts, so that we reach the given precision/accuracy and scope. We then also get the additional trade-off between how fast we increase the granularity, because if we increase it in large steps, we will faster reach the required level of precision, but may also use too much effort. However, with small increments, we may risk of using many iterations of the method.

2 Stakeholders

In this section we identify the stakeholders that are interested in results of the CloudScale project so they can apply them in providing scalable services in the cloud environment. These stakeholders use CloudScale results as either service providers or consumers, either as the developers of the services in the cloud (on infrastructure, platform, or software level). The stakeholder roles that are applicable for various cloud stack layers are marked with the XaaS ("something as a service") term.

2.1 Service Consumer

Service consumer is the person or enterprise entity that uses the service and its resources. Service Con-
sumer wants to have supplied services for his own purpose according to business needs and according to an agreed SLA. Since there are different delivery models of cloud services (Infrastructure as a Service, Platform as a Service, Software as a Service and all of their subgroups) we can distinguish different service consumer types for each of these delivery models. Depending on the type of service and their role, the consumer works with different user interfaces and programming interfaces.

Based on the service consumer type we can abstract several service interaction types:

- Application usage
- Administrative functions - start/stop virtual machine (VM), manage cloud storage, system configuration
- Programming interfaces for application development

2.2 Service Provider

The service provider delivers the service to the consumer. Service provider is responsible to fulfil SLA and other requirements towards service user (according to cloud services, service provider can be IaaS, PaaS or SaaS provider), prepare service requirements and interact with system builder to enable appropriate service, operate system during system life-cycle and lead needs for system adaptation.

2.3 Service Developer

The service developer develops a cloud service for a deployment model on a certain layer in the cloud stack. Service developer is responsible for service realisation (both development and test), and for preparing the system deployment process. This role cooperates with System architect for checking realized services and with System engineer in preparing system deployment.

Most of the current deployed cloud services are developed for SaaS cloud deployment model. Services developed for the IaaS and PaaS deployment models will subsequently be used by SaaS developers and cloud providers. Service developer uses infrastructure, as well as all accompanying mechanisms, provided by the cloud service provider on the certain cloud stack layer.

2.4 Product Manager

Product manager is responsible for identifying initial system requirements and defining development goals especially from the business perspective. The product manager is engaged in making decisions regarding the requirements fulfilment and business potential of the solution.

The main interests of the product manager are the overall system behaviour, architecture compliance and price of the final solution. He is also a final decision maker for the solution and negotiation with customer about service/system acceptance and approval of system evolution during runtime phase.

2.5 System Architect

System architect is the architect of a certain layer in the cloud stack and the main system modeller. He is responsible for selection of the system components on a certain cloud layer and their interaction. Responsibility of the system architect is to find an optimal cloud service organization and deployment for produced cloud services. During this process System architect cooperates with Product manager and Service developer. From CloudScale tools point of view, he is also the main user of the CloudScale tools and methods during design project phase. During system design phase system architect needs to provide optimal evolution scenario and include optimal system components into system architecture (related to How To CloudScale result).

2.6 System Engineer

System engineer is person responsible for service deployment and monitoring of the system in operation. Based on monitoring results, the system engineer tends to optimize system operation parameters or to run the System Construction and Analysis process step if it is not possible to fix system by fine tuning. System engineer cooperates with all other roles during the system life-cycle.

System engineer is responsible for the runtime system monitoring process and identification of the system critical elements (regarding scalability and performance issues). This role is the initiator of the system evolution phase after fine tuning system adoption phase.

3 Overall CloudScale Method

Figure 1 gives a high level overview of the CloudScale Method. The figure shows how decisions guide the user through tool-driven process steps. Some of the steps are decomposed in more detailed method steps. Before describing this high level method, our graphical notation, described in the
The legend of this figure, is described in Section 3.1. The steps in this overall method are described in Section 3.2.

Our method extends the Q-ImPrESS method [11], building on the Palladio performance modelling tool [3]. A first draft of this method was introduced in [4].

3.1 Notation

The notation used in Figure 1 as well as its decomposed method steps in later figures are as follows:

- **Start or stop:** describes the start and stop of these method steps. For a decomposed process, we start and stop where the high level process starts and stops.
- **Tool-driven decomposed process:** a tool-driven process which is described in more detailed in later process diagrams.
- **Tool-driven process:** a task which is not described further.
- **Document:** a file used to store text or models.
- **Manual tasks:** manual, complex tasks which may also be assisted by tools.
- **Decision:** which may also involve complex manual tasks.
- **Parallel tasks:** synchronisation points between or after parallel tasks.
- **Data flow:** some sort of data in the specified direction.
- **Data & control flow:** in addition to data also manual work.

3.2 Steps in Overall Method

System architects first need to define coarse requirements for scalability, elasticity and efficiency. These requirements are input to the CloudScale Method and tools and may be refined during the analysis. The first decision in the diagram, branches based on the type of available artefacts. If code is already available, we are in a reengineering scenario. In case no code is available or it should not be used, analyses can be based on an architectural model in a forward engineering scenario. In reengineering scenarios the analysis may be based either on the static code or on the running application in the Spotters process step. The Spotters consist of two corresponding tools: The Static Spotter examines static code while the Dynamic Spotter comprizes instrumentation and load generation of a running service. In both cases, the code is reengineered if anti-patterns or root causes are spotted.

Alternatively, the analysis can be based on modelling the system's architecture, its usage by users, and its need for hardware and software resources. The resulting model is persisted in a special architectural modelling language, called ScaleDL. ScaleDL is comparable to an UML2 model with MARTE quality annotations, e.g., resource demands or workloads. The model can be either a complete new design, a refinement of an earlier system model, or a design which is partially extracted from existing implementations via the Extractor reverse engineering tool. ScaleDL models contain specifications of various application aspects ranging from the implementations structure and behaviour, over the systems resource demands, to the systems usage and its change over time as well as a description of the autonomous elasticity manager.

After realising a service the Static Spotter may be used on the code. After deployment we may use Dynamic Spotter on the deployed service. If new requirement violations arise, a new iteration of the CloudScale Method needs to be executed.

Once the selected analysis (based on the Spotters or the Analyzer) indicates that scalability, elasticity, and cost-efficiency requirements are sufficiently met, the system has to be realized, deployed, and operated next. For new systems, system engineers realize the system based on the ScaleDL model, i.e., based on the system's architectural representation. For existing systems, system engineers semi-automatically reengineer detected issues based on either Spotter or Analyzer suggestions.

After the so-realized system is deployed, engineers can operate the system and test whether their requirements are indeed met. If engineers detect requirement violations at this stage, they can iterate through the CloudScale Method to improve the situation, again supported by dedicated detection tools. If the system meets its requirements, engineers can stop and only have to reenter the CloudScale Method in case requirements or the system's context change.

4 Typical Scenarios

System engineers will choose different paths through the CloudScale Method depending on which concrete scenario they want to cope with. In this section, we therefore describe the paths through the CloudScale Method for the most common scenarios: analyzing a (new) system (Sec. 4.1), migrating an existing system (Sec. 4.2), and analys-
Figure 1: Overall CloudScale Method steps.
4.1 Analyzing a (new) System

When system engineers develop a system from scratch, they want to know its quality properties as early as possible. The CloudScale Method therefore provides a quality analysis that already works on the architectural model (ScaleDL model) of the system, i.e., before any source code is available. For existing systems with source code available, such an analysis can also be conducted; a part of the ScaleDL model can automatically be extracted in this case. System engineers execute the following actions of the CloudScale Method. They want to conduct a model-driven quality analysis (first decision node in Fig. 1) and therefore have to decide whether to extract a partial ScaleDL model or to manually specify a ScaleDL model using CloudScale’s modeling tools (second decision node in Fig. 1). If system engineers cope with a new system or want to update an existing ScaleDL model, they directly use CloudScale’s modeling tools (details on ScaleDL modeling in Sec. 7). If engineers have source code but no ScaleDL model available, they first extract a partial ScaleDL model directly from code before completing the ScaleDL model in the next step (details on ScaleDL extracting in Sec. 5). Afterward some manual steps, system engineers can use their ScaleDL model as input to the Analyzer. The Analyzer simulates whether scalability, elasticity, and cost-efficiency requirements are sufficiently achieved, thus, allowing system engineers to iteratively improve the architectural model until satisfied (details in Sec. 8). Once satisfied, system engineers can realize, deploy, and operate the planned system with a lowered risk of violating requirements (details in Sec. 3). If system engineers — during testing or operation — still detect requirement violations, they can iterate through the CloudScale Method. Because the system has now been realized, system engineers can decide for any of the typical scenarios.

4.2 Migrating an Existing System

The cloud computing paradigm promises advantages for scalability, e.g., virtually unlimited computing resources. However, migrating to a cloud computing environment does not automatically guarantee scalability. Therefore, when system engineers want to move an existing system to a cloud computing environment, they have to analyze whether their system fulfills scalability requirements or suffers from scalability issues. With the CloudScale Method, system engineers are able to use scalability detection tools, i.e., the Static and the Dynamic Spotter. The Static Spotter detects potential scalability issues at the code and model level. These potential scalability issues point the Dynamic Spotter to the code it should instrument. Based on a workload generation, Dynamic Spotter subsequently detects whether potential scalability issues manifest as actual scalability issues. After software engineers eliminate detected scalability issues, the existing system becomes more scalable, thus, being able to benefit from cloud computing environments advantages.

4.3 Analyzing a Running System

Often system engineers have an already running, legacy system they plan to migrate to cloud computing environments. In this context, we can utilize the running application and perform performance and scalability tests on it. In Figure 1 this scenario is covered by taking the left branch of the Analysis based on code decision node. In addition to the code, engineers need to provision a representative infrastructure environment and install the application in it. Furthermore, they need to provide realistic load scripts, e.g., using Apaches JMeter. Finally, engineers need to specify their applications requirements as SLOs. Using these inputs, we can use a dynamic spotter approach which basically test drives the application in different work and load situations and identifies whenever the application does not scale in the performed tests. Based on a detailed analysis which of the tests failed and for what reason they failed, the dynamic spotter reports a set of diagnostic statements reporting found scalability anti-patterns (also called HowNotTos in CloudScale’s terminology). Engineers can then systematically focus on the found issues and resolve them, paving the route to a successful cloud computing setup. Once the dynamic spotter does not find any more SLO violations, engineers can continue to install their application in the new environment. In case not all issues can be easily resolved, engineers need to reengineer their applications by taking the look in the CloudScale process originating in the Requirements met decision node.

5 Extractor Method Steps

The Extractor is a reverse engineering tool for automatic extraction of ScaleDL models, thus, lowering modeling effort for system engineers if source code already exists. Extractor is based on the Archimetrix approach [10] that combines different reverse engineering approaches to iteratively
recover and reengineer component-based software architectures. Accordingly, as shown in Figure 2, the input to the Extractor is source code and configuration parameters for reverse engineering, e.g., thresholds that specify when to cluster classes into components. First we have to decide which part of the system should be extracted. In a large system we may only be interested in a few critical services. After parsing the source code, Extractor clusters relevant elements based on these parameters into software components. The output is a partial Extended PCM model, i.e., a model that covers the component-based structure of the extracted source code as well as its control and data flow. The model is partial because it misses context information like system usage and hardware specifications.

![Figure 2: Extractor method steps.](image)

While Extractor relieves system engineers from potentially tedious modeling tasks, engineers have some effort to find the right configuration parameters. In general, engineers start with Extractor’s default configuration and assess whether the resulting Extended PCM model is satisfying for their system. Engineers are typically unsatisfied if the result is too abstract (e.g., Extractor clustered the whole system into one component) or too fine-grained (e.g., Extractor clustered each class into a dedicated component). In that case, system engineers alter configuration parameters and rerun Extractor until satisfied. The main parameters for Extractor [5, p. 16] are (default values included):

- Clustering Merge Threshold Min (Start Value) (45)
- Clustering Merge Threshold Max (End Value) (100)
- Clustering Merge Threshold Increment (10)
- Clustering Composition Threshold Min (End Value) (25)
- Clustering Composition Threshold Max (Start Value) (100)
- Clustering Composition Threshold Decrement (10)

In addition, the Extractor also has more values (also with default values):

- Merge: Interface Violation (10)
- Composition: Interface Adherence (40)
- Package Mapping (70)
- Directory Mapping (70)
- Highest Name Resemblance (45)
- High Name Resemblance (30)
- Mid Name Resemblance (15)
- Low Name Resemblance (15)
- High SLAQ (Slice Layer Architecture Quality) (15)
- Low SLAQ (5)
- DMS (Distance from the Main Sequence) (5)
- High Coupling (1)
- Low Coupling (0)

6 Spotters Method Steps

The decomposed method steps for the Spotters are shown in Figure 3. There are two Spotters. The Static Spotter is an automated scalability anti-pattern detection tool based on a static code analysis. The Dynamic Spotter is an automated scalability anti-pattern detection framework based on measurements, thus, giving system engineers an additional mean to lower detection efforts. The Dynamic Spotter can in addition to finding anti-patterns also test scalability. In the next sections we describe each of the Spotters in more detail, before we elaborate on the decision about satisfaction.

6.1 Static Spotter

Static Spotter is based on the Reclipse [12] tool that detects patterns within source code structures. The input to Static Spotter is source code decorated models together with a scalability anti-pattern catalogue. These models are a result of the Extractor tool (See Sec. 5). Therefore, unless such models are available from previous uses of the Extractor, then
the Extractor has to be performed with a suitable set of configuration parameters. The anti-patterns catalogue in CloudScale for example includes the "One Lane Bridge" anti-pattern, i.e., a pattern to detect synchronized blocks that potentially become a bottleneck for increased workloads. After searching for such anti-patterns, Static Spotter outputs a list of found anti-pattern candidates. System engineers have finally to assess whether these candidates indeed manifest in scalability issues and resolve identified scalability issues.

As the Static Spotter runs automatically, system engineers have little effort to detect potential scalability anti-pattern candidates, especially if they reuse CloudScale’s anti-pattern catalogue. The main effort is to identify the candidates that indeed manifest in scalability issues and to resolve such issues. The effort for such an identification and resolution heavily varies depending on the concrete system. System engineers can either conduct these tasks manually (e.g., by specifying system-specific test cases for candidates) or reduce manual efforts by directing the Dynamic Spotter to inspect anti-pattern candidates (see next section).

6.2 Dynamic Spotter

Dynamic Spotter systematically identifies scalability issues by generating load to a system in operation and diagnoses their root causes [14]. Before we start Dynamic Spotter we must fulfill the following preconditions:

- An adaptor for load generator must exist, for example to Apache JMeter, HP LoadRunner or Gatling.
- An adaptor for instrumentation and measuring must also exist, for example to DiSL, Kieker or AIM.
- Configure a load generator script with ramp up and cool down time as well as a probability distribution between the operations.
- Service must be started with an instrumentation agent.

After starting Dynamic Spotter we must set these configuration parameters:
• Quality threshold (for all operations, not for individual operations): what is the meaning of this?
• Percentile for Quality Thresholds (QTs), for example 10% of invocations can be violate QT.
• Duration of ramp up and cool down.
• Run time with stable load.
• Maximum load.
• Load increment for each run.
• Which anti pattern to focus on: App Hiccups, Blob (or God class), Continuous Violation, DB Congestion, EDC (Expensive Database Call), Empty Semi Trucks, Excessive Messaging, OLB (One Line Bridge), Perf Problem, Ramp, Static Spotter Synchronized Method, Stifle or Traffic Jam.
• Dynamic Spotter scope (only applies to One Line Bridge), which can then be:
  – entry point: top level operation of the service; for a servlet framework this will not give any details as there will only be one entry point
  – database: will know the exact query, but will do not know the classes or methods calling this query
  – middle: there are also a potential for getting a middle level, i.e. identify which packages, classes or methods that are called.

The statistical precision on applying Dynamic Spotter is increased when we increase:
• ramp up time (will not by itself increase precision, but the service under test will be put under less stress with a longer ramp up time, and in this sense the statistical precision will be increased).
• measurement duration (duration with a stable load, between ramp up and cool down).
• # load levels. Each load level corresponds to a arrival rate (open) or to a given number of user (closed).

With a broader scope in OLB we can be more certain to find a scalability problem, but it will not be easy to spot where in the code it is.

After searching for anti-patterns, Dynamic Spotter outputs found scalability issues. If you have found anti-patterns, a more narrow search may be beneficial. If no anti-patterns are found, you may search better by increasing the measurement time or by using higher load levels. For a person with knowledge of the service under test, this will be valuable information, even it the exact location of the scalability problem cannot be identified precisely.

In contrast to the Static Spotter, system engineers can directly identify manifested scalability issues (i.e., not only candidates), but have more effort to instrument and operate an existing system. Once such a system is available, system engineers have only little effort to detect scalability issues, especially if they reuse CloudScale’s anti-pattern catalogue.

Run time is determined by number of load levels (total load, as well as initial load and load increments) as well as total run time for each load level (ramp up, stable load, and cool down time). Depending on the concrete system, Dynamic Spotter may run for a long while, e.g., during a whole night. To alleviate this overhead, Dynamic Spotter can optionally take the anti-pattern candidates identified by the Static Spotter as an optional input. These candidates direct Dynamic Spotter to instrument the most relevant parts of a system, thus, reducing runtime.

7 ScaleDL Method Steps

ScaleDL is an architecture description language that system engineers can use to document and analyze their systems regarding scalability, elasticity, and cost-efficiency. ScaleDL covers structure and behavior of the system as well as contextual factors like the concrete cloud computing environment and the behavior of system users over time. An optional input to the ScaleDL specification step is an exiting ScaleDL model to be altered or extended, e.g., received from the Extractor (Sec. 5). For editing existing or for specifying new ScaleDL models, engineers use the editors of CloudScale’s integrated development environment, the so-called CloudScale Environment. The decomposed method steps for modelling ScaleDL models are shown in Figure 4.

The effort for modeling tasks varies depending on the complexity of the system. For a medium-size online shop, we engineered a ScaleDL model within one person month. This effort pays out if scalability, elasticity, and cost-efficiency are crucial and need an early analysis based on analyzing the ScaleDL model with the Analyzer in Section 8.

ScaleDL consists of the five sub languages: Extended PCM ScaleDL Overview, ScaleDL Architectural Templates, ScaleDL Usage Evolution, and
DLIM. These five sub models are described individually in the following sections.

7.1 Extended PCM

Extended PCM allows architects to model the internals of the services: components, components’ assembly to a system, hardware resources, and components’ allocation to these resources. The extension allows additionally to model self-adaptation: monitoring specifications and adaptation rules. We will first describe basic PCM in Section 7.1.1 and afterwards the extensions in Section 7.1.2. Both for basic PCM as well as the extensions we will focus on the manual tasks.

7.1.1 Basic PCM

The Palladio [3] approach comes with a model to specify component-based systems, the Palladio Component Model (PCM). The PCM allows architects to model components, assemblies of components into systems, hardware resources, the allocation of components to these resources, and static usage scenarios. These are standard performance engineering concepts and not unique to CloudScale, which as means that the manual tasks are standard performance engineering steps like making a resource environment, an allocation model, and or a usage scenario. Also identifying components an in particular measuring resource demands are major manual tasks.

The Palladio approach also comes with a tool, the Palladio-Bench, allowing to analyze a system (i.e., a PCM instance) regarding quality properties. The Palladio-Bench supports performance (response time, utilization, throughput) as well as safety (mean time to failure) quality properties. Features of the basic PCM may be specified using ScaleDL Overview but also directly in Extended PCM.

Generally, a detailed PCM will require many parameters, whereas a coarse model only requires some. In particular, granularity is determined by the 1) service itself and its internal components, 2) its usage, 3) its resources as well 4) allocation of internal components to resources. We will describe the granularity of each of these levels in turn.

Granularity of the service itself:

- # components: on the one extreme each method is a component, and on the other ex-
treme the complete service only consists of one component. In between we can have one component for each class, or we may put classes together.

- # operations in each component: a detailed characterisation of a component has many operations, whereas a coarse characterisation has few or only one operation.

- # connections between the components: a coarse characterisation ignores less important connections, whereas a detailed characterisation includes all possible connections.

Granularity of the usage of the service:

- # user operations: representing the way users invoke the service. If the user mainly use a few operations, a coarse characterisation may ignore rare operations. However, rare operations may be important because they may have large resource demands.

- # work parameters in the user operations: few or many work parameters for the operations and in each operation.

- Probabilities of operations: spanning from a full Markov matrix to only probabilities of each operation (a vector).

- # user scenarios like the browsing mix or the shopping mix.

- Quality thresholds (QTs or SLOs – Service Level Objectives): We must specify 1) Where to measure (usage scenario, active/passive resources, operations etc.) 2) What to measure (90 percentile response times, average response times, utilisation, throughput, etc.) 3) Thresholds (fixed, fuzzy, upper/lower bounds).

Granularity of the resources:

- # servers: may ignore some servers if they are not important, for example for payment or for images.

- # resources: may ignore network and disk and only focus on processing resources like CPUs.

- # operations on resources: discriminate between store and retrieve for a disk.

- Work modelling of operations, spanning from work parameters like message lengths to amount of disk storage.

- Statistical distribution of resource demands or only a constant. The sophistication of the statistical resource demand distribution may also vary, from a coarse model with only two different values where one probability is enough to describe the distribution between them, to many different resource demand values where several probability values are required to characterize their distribution.

Granularity of the allocation of components to resources:

- Many links between components and resources or fewer links.

Generally a more detailed PCM model will require more manual work. On the other hand it may be more straightforward and therefore also easier to make a more detailed PCM model. A more detailed model will more accurately resemble the actual service. This is also the case when introducing more sophisticated statistical resource modelling. On the other hand a model with no statistical modelling of resources (i.e. where only one value is used for resource demands and not many potential values distinguished by a probability) will have a lower confidence intervals, so in this case accuracy will be improved with more detailed resource modelling, whereas precision will be reduced.

7.1.2 PCM Extensions

Simulizar [1] extends Palladio to support modeling and analysis of self-adaptations. The concept of self-adaptation is important for Cloud-Scale because we want to analyze elastic cloud computing environments, e.g., systems that can scale-out and scale-in based on currently monitored load. Simulizar’s concepts of self-adaptation allow to model and analyze such systems. For this, Simulizar enriches the PCM to specify (1) monitoring annotations and (2) adaptation rules. Monitoring annotations allow to mark PCM elements, e.g., an operation of a component, to be monitored during analysis using a metric such as response time. Adaptation rules can react on changes of monitored values. For example, when a certain response time threshold is exceeded, an adaptation rule could trigger a scaling out of bottleneck components. Simulizar allows to consider these adaptation rules during system analysis. We also have granularity trade-offs for these PCM extensions.

7.2 ScaleDL Overview

The ScaleDL Overview is a meta-model that provides a design-oriented modelling language for
7.3 ScaleDL Architectural Templates

ScaleDL Architectural Templates allows architects to model systems based on best practices as well as to reuse scalability models specified by architectural template engineers. Engineers are guided along CloudScale’s best practice templates (Architectural Templates [6]) for modeling cloud-based systems. For example, the “Dynamic Horizontal Scaling” template allows engineers to model horizontal scaling, i.e., a system can dynamically replicate itself over additional cloud computing resources to cope with increasing workloads.

How to create ATs is not described in this CloudScale Method, but we must say that granularity trade-offs when making ATs is simply the same as when making PCM models. Therefore, see Section 7.1.1 for a detailed description of the granularity trade-offs for modelling ATs. Also, the ATs you model, the easier it will be for developers, because they will then use considerably less manual work.

7.4 ScaleDL Usage Evolution

Before we describe ScaleDL Usage Evolution, we need to introduce work and load. Work characterizes the amount of data to be processed, stored or communicated by a service. A service may have none or several work parameters.

A service has one or more operations. The probability of each of these operations is described by the operation mix. Load describes how often the operations in a service is invoked. On average the probabilities the operations follow the operating mix. With a constant number of users, we have a closed system which is specified by the number of users (N), in addition to the think time (Z). A variable number of users is an open system. For an open system we use arrival rate, \( \lambda \). A quality metric defines how we measure a certain quality and is a key part of an SLO (Service Level Objective). Quality thresholds (QTs) describe the border between acceptable and non-acceptable quality. Generally there may be one or more quality metrics for each operation as well as a quality threshold corresponding to each quality metric and operation.

Usage evolution describes how a parameters change as a function of time. In CloudScale we focus on evolution of load, but we also handle evolution of work. Quality threshold evolution is further work. To realize usage evolution in the Analyzer, we combine the usage model in Palladio with DLIM.

Usage evolution granularity are determined by:

- # operations with independent load evolution.
- # work parameters with independent work evolution.
- # usage scenarios, if we change operation mix: probabilities of operations as well as quality thresholds.

7.5 DLIM

Descartes Load Intensity Meta-Model used by the load intensity modelling tool LIMBO [13]. Using LIMBO we can then describe the evolution of load and work parameters in the following way: seasonal (daily, weekly, monthly or yearly variations), trend variations (typically linear growth) or noise. This characterization has granularity trade-offs concerning the level of detail in the evolution, for base, seasonal (# peaks, # periods: day, week, month, year), trend, bursts, noise. A detailed model will have fluctuations in several time axes, but it the evolution for each of the time axes can also be more detailed with a richer set of values. The level of detail may be higher for load, compared to work evolution. For scalability, we may focus on the highest load and work as well as on the strictest quality thresholds. For efficiency much are details are required.

Usage evolution may also be measured and not only modelled.

Usage evolution together with allocation will essentially be scalability, elasticity as well as efficiency requirements for a service.

8 Analyzer Method Steps

The Analyzer is a simulator for ScaleDL models. During simulation, Analyzer measures typical cloud computing properties, i.e., scalability, elasticity, and cost-efficiency. Based on these measurements, system engineers can decide whether their service requirements described in Figure 1 are met. If these requirements are violated, system engineers can iteratively alter their ScaleDL model and
check whether this alteration improved the situation. Once requirements are met, system engineers can continue to realize, deploy, and operate their service — with a reduced risk that their system violates requirements.

Figure 5 describes the decomposed process steps for the Analyzer. Before starting to analyze a ScaleDL model, we must determine which metrics to investigate and also set other Analyzer configuration parameters. The implementation of Analyzer is based on existing simulators for performance (Palladio [3] and SimuLizar [1]). Therefore, Analyzer can simulate typical performance metrics like response times, throughput, and utilization as well.

Figure 5: Analyzer method steps.

In CloudScale, we integrated metrics for scalability, elasticity, and cost-efficiency [7]. Our integration is based on a systematic literature review of definitions and metrics for these properties [8] and a systematic derivation of further metrics [2]. For example, the number of service level objective violations [2] elasticity metric counts the number of violated performance requirements during adaptation phases. Another example is the mean time to quality repair [2] elasticity metric that measures the time a system needs to move from a state that violates performance requirements to a state that satisfies all performance requirements. An example for a cost-efficiency metric is the cost over time [8] metric that computes the operation costs accrued for using cloud computing resources per billing interval.

Such metrics enable system engineers to conduct according trade-off analyses, e.g., service level objectives violations vs. costs. The benefit of the Analyzer is that such analyses are already enabled at design-time, i.e., before developers actually implement a system following the architectural borders of the architectural model. The main effort for using the Analyzer is the preceding specification of a ScaleDL model. In our experience, the beneficial measurements of the Analyzer outweigh this effort.

The Analyzer step in this decomposed process covers all the manual and automated steps required to get the required metrics, once the input data are present. Looking at the metric results, the user must consider if he is satisfied with the result of the simulation. If not, the ScaleDL model described in Section 7 is modified and a new simulation is performed. Typical aspects of modify are the following aspects:

- improve service model (as expressed by Extended PCM). This will also include usage scenarios: like number of operations as well as their probabilities.
- relax usage evolution, so that peaks are lower, for example.
- modify resources: typically using stronger or more CPUs.
- modify allocation.

The Analyzer configuration variables determines run length as well as both precision and accuracy:

- Run length: determines the accuracy of the results. A longer run length will typically give a higher confidence intervals, but then also a better match to the "real" service.
- Number of times to run the model (with different seeds):. More runs will also give a higher confidence interval and therefore seemingly worse confidence intervals, but then with improved accuracy.
- Which metric to use: A robust metric can tolerate larger variations compared to a more sensitive metric.

Acknowledgement

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant no 317704 (CloudScale).

References


