Chapter 5
The CloudScale Method

Gunnar Brataas and Steffen Becker

Abstract This chapter details the CloudScale method. We describe its high-level process with the most important steps. We look more closely at the CloudScale method from Sect. 2.1 and detail it with respect to the developer roles executing it. We also introduce the two major method use cases. Method use case I is about analyzing a modeled system; method use case II deals with analyzing and migrating an implemented system. All discussions in this chapter are guided by the granularity of the analysis you want to perform, hence; this chapter also introduces granularity as a key concept and discusses how to find the right one.

As granularity is important for all steps of the CloudScale method, it is introduced in Sect. 5.2. As a second basis, our graphical notation is described in Sect. 5.3. The method description starts with an introduction into the CloudScale method roles in Sect. 5.4. As a core section in this chapter, Sect. 5.5 gives a detailed overall overview on the CloudScale method. Afterward, the following sections give details on all method steps: Sect. 5.6 outlines how to identify service-level objectives (SLOs), critical use cases, and their associated key scenarios from business needs; Sect. 5.7 then describes how to transform the SLOs and critical use cases into scalability, elasticity, and cost-efficiency requirements. Afterward, the two main use cases of the CloudScale method are introduced: Sect. 5.8 outlines how to use models to analyze a system's properties, while Sect. 5.9 sketches how to analyze implemented and executable systems. Finally, Sect. 5.10 briefly describes how to realize and operate the system.

G. Brataas (✉)
SINTEF Digital, Strindvegen 4, 7034 Trondheim, Norway
e-mail: gunnar.brataas@sintef.no

S. Becker
University of Stuttgart, Universitätsstraße 38, 70569 Stuttgart, Germany
e-mail: steffen.becker@informatik.uni-stuttgart.de

S. Becker et al. (eds.), *Engineering Scalable, Elastic, and Cost-Efficient Cloud Computing Applications*, DOI 10.1007/978-3-319-54286-7_5

5.1 Introduction

As already described in Sect. 1.10, the CloudScale method guides stakeholders by describing the steps to follow when engineering scalable, elastic, and cost-efficient systems. It also describes the set of different stakeholders involved. The CloudScale method as presented in this book is based on initial ideas published in [1] and further refined in [2]. In addition, it was inspired by the Q-ImPrESS method to engineer evolving service-oriented systems [3].

The basis for the CloudScale method is CloudScale's tools, which automate some parts of the method. The CloudScale method describes the input as well as the output of the CloudScale tools. In some cases, the input to a CloudScale tool is produced by another CloudScale tool, but manual steps may also be required to produce the required inputs.

Before starting the CloudScale method, you must have a clear idea of what you want to learn. Possible overall objectives are:

- Trade-off between cost, functionality, and quality during development. You may, for example, compare two different architectures.
- Trade-off between cost, functionality, and quality during modification. The new parts can be a new operation, or a new architecture. A new architecture may simply mean to compose existing services and components differently, but it may also mean replacing some of them.
- Compare scalability, elasticity, and cost-efficiency of competing services.
- Compare competing deployments for an existing service.

Common for all these objectives is that you need to do some sort of scalability, elasticity, or cost-efficiency analysis. The CloudScale method is a method for performing such analysis. When you do this analysis, you 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. An important part of this method is the necessary manual steps, like setting objectives for scope and accuracy, setting configuration parameters, instrument source code, or finding out if the defined quality objectives are met. Which guidance is available for performing them and how do you know if you have to adjust something, and then what shall you adjust?

5.2 Granularity

By granularity, we refer to the 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 you put into the analysis as well as what you get out of the analysis and the time an analysis will take.

For what you 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. Making 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 increases when you model more operations, more components, more resources, and more complex relation between them.
- Run time to do the analysis on the computer. Most often this time will be negligible, but for some analyses you may spend several hours, and then this time will also influence 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 you 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 [4]. To increase precision (i.e. reducing confidence intervals), you can run a simulation several times (with different seeds).
- Accuracy, the difference between the reported value and the "real" value [4]. Generally, a more complex model may be more accurate, since it is then easier to match reality. Validation where you compare the "actual" service with the modeled service is the key to increase accuracy [4].
- Scope, relating to coverage. You may look for scalability problems in the complete service as well as all its underlying services, a broad scope, but you may also confine the scalability investigation to one of the many classes inside of the service, a narrow scope. Similarly, you may focus on the use of processing resources and ignore the use of storage resources. You 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 comprehensive 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 afterward becomes: which precision, accuracy, and scope are required for spotting scalability/elasticity/cost-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 cost-efficiency. We recommend a coarse approach first and afterward to increase the granularity of key parts so that you reach the given precision/accuracy and scope. We advise to avoid small increments as this leads to too many iterations of the method.

5.3 Method Notation

The notation used in this chapter as well as its decomposed method steps in later chapters is shown in Fig. 5.1 and explained below:

- Start or stop: describes the start and stop of these method steps. For a decomposed process, you start and stop where the high-level process starts and stops.
- Tool-driven process: a task which is supported by a tool. This process may be later decomposed.
- Decision: the flow of control depends upon a decision. This decision may involve complex manual tasks.
- Artifact: a file used to store text or models.
- External artifact: an artifact which is external to the process shown.
- Manual tasks: manual, complex tasks which may also be assisted by tools outside of CloudScale like text editors, compilers, monitoring tools, etc.
- External manual task: a manual task which is external to the process shown.
- Role: one of CloudScale's roles.
- Data flow: data flow in the specified direction.
- Data & control flow: data flow as well as flow of control with manual work in the specified direction.
- Parallel tasks: synchronization points between or after parallel tasks.

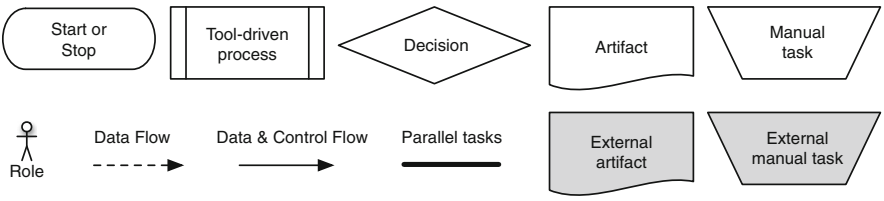

Fig. 5.1 Notation used in the CloudScale Method

5.4 Roles in the Method

In this section, we identify major roles required to (re)design scalable services in a cloud environment. We define the following six roles:

Service consumer: a person or enterprise entity that uses the service and its resources and therefore has a service-level objective (SLO) for this service. The service consumer is part of a larger business process. As this business process has business needs, there are resulting SLOs the service consumer needs to have fulfilled by a supporting IT system. Since there are different delivery models of cloud services (Infrastructure as a Service [Iaas], Platform as a Service [PaaS], Software as a Service [SaaS], and all of their subgroups), you can distinguish different service consumer roles 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.

Product manager: responsible for identifying system requirements and defining development goals, especially from the business perspective. The product manager is engaged in making decisions regarding the requirements fulfillment and business potential of the solution. The main interests of the product manager are the overall system behavior, architecture compliance, and price of the final solution. He is also a final decision-maker for the solution and negotiation with the customer about service/system acceptance and approval of the system evolution during operation.

System engineer: responsible for deploying the service and for the monitoring of the system in operation. Based on monitoring results, the system engineer optimizes the system's operation parameters. If required, the system engineer initiates the system evolution process steps so that further redesign and reimplementation of the system may be triggered if it is impossible to fix the system by fine tuning. The system engineer cooperates with all other roles during the system lifecycle.

System architect: is the architect of a certain layer in the cloud stack and the main system modeler. He is responsible for selecting the system components on a certain cloud layer and their interaction. The responsibility of the system architect is to find an optimal cloud service organization and deployment for all used cloud services. The system architect cooperates with the product manager and the service developer. He is also the main user of the tools and methods during design. During the system design phase, the system architect needs to provide an optimal evolution scenario and include optimal system components into the system architecture.

Service developer: develops a cloud service for a deployment model on a certain layer in the cloud stack. The service developer is responsible for both development and testing during service realization, and for preparing the system deployment process. The service developer cooperates with the system architect for checking realized services and with the system engineer when preparing system deployments. Most of the current deployed cloud services are developed

for SaaS cloud deployment models. Services developed for the IaaS and PaaS deployment models will subsequently be used by SaaS developers and cloud providers. The service developer uses infrastructure, as well as all accompanying mechanisms, provided by the cloud service provider on certain cloud stack layers.

Service provider: delivers the service to the consumer. The service provider is responsible for fulfilling SLOs and other requirements toward service consumers. She prepares service requirements and interacts with system engineers to enable an appropriate service, operates the system during the whole system lifecycle, and identifies the needs for system evolution.

The cloud computing definition published by the NIST [5] defines five basic cloud computing actors: cloud consumer, cloud provider, cloud auditor, cloud broker, and cloud carrier. The NIST actor cloud consumer is similar to the service consumer role in CloudScale. However, we focus more on the service itself and less on the cloud infrastructure. The latter three roles may also be relevant to scalability analysis, but are not considered further in this book.

The NIST actor for cloud providers is further decomposed into five NIST actors for service deployment, service orchestration, cloud service management, security, and privacy. The CloudScale role of system engineer aggregates the NIST actors for service deployment, service orchestration, cloud service management, but with a focus on scalability engineering. The NIST actors for security and privacy are complementary to the CloudScale roles.

The CloudScale roles for product manager, system architect, service developer, and service provider are not explicitly mentioned by the NIST, but they are relevant to CloudScale with our focus on (re)design.

5.5 Method Steps

The CloudScale method was introduced in Sect. 2.1. In this chapter, we show which roles are relevant to each method step. We also point out two main method use cases: one for modeling projected services and the other for reengineering existing services.

The product manager has tight cooperation with the service customer, and as a result, the product manager identifies the need for a new service. He or she elicits more or less vague business-oriented scalability, elasticity, and cost-efficiency requirements for this new service. In cooperation with the service provider, the product manager identifies SLOs, critical use cases, and key scenarios. In our context, critical use cases are the riskiest operations. Key scenarios describe when these operations have the highest workload.

Based on the business-oriented scalability, elasticity, and cost-efficiency requirements defined in the previous step, the system architect joins the discussion with the product manager and the service provider, and, together with them, agrees

on projected workloads. In this way, the technical requirements for scalability, elasticity, and cost-efficiency requirements are established.

In this scenario, the system architect has to design a new system. Therefore, she uses the CloudScale method in method use case I: the analyses will be based on a model as no useable source code exists so far. This method use case is described in more detail in Sect. 5.8. In this method use case, the system architect, together with the service developer, specifies a Scalability Description Language (ScaleDL) model. This is a complex task with several manual steps, where the system architect provides high-level information concerning the overall architecture describing how the components fit together and where the service developer fills in all the details about the components. Some details may be required from the system engineer concerning the cloud resources used. Moreover, the requirements for scalability, elasticity, and cost-efficiency are detailed as part of working with the ScaleDL model.

When the system architect, together with the service developer, is satisfied with the ScaleDL model, it is fed into the Analyzer. The system architect, together with the service developer, uses the results from the simulation in the Analyzer to improve both the high-level architecture as well as more detailed design choices. When the service developer, together with the system architect, decides that the service model is sufficiently mature, it is realized. After realization, service implementation is later deployed.

When a service is deployed, the system engineer takes over the responsibility. Hopefully, he will discover any poor scalable, elastic, or cost-efficient behavior before the service consumers do, and will then trigger the required reengineering steps in method use case II. Method use case II deals with analyzing and migrating an implemented system, and its steps are performed using the Spotter tools, which can identify scalability root causes based on either code or a running system. This method use case is described in more detail in Sect. 5.9.

There are fundamentally two ways of progress during design: adding details and making implementation decisions. These two will often be linked. As part of the development process, when detailing the architecture or implementing the system, the level of detail will increase:

Decomposition of operations: so that one operation becomes several operations. The opposite process of aggregation of operations, where operations are merged, is less likely.

SLOs may be detailed: where groups of operations sharing an SLO each get individual SLOs.

More work parameters: may be required to cope with the increased level of detail in a more elaborate design.

5.6 Identify Service-Level Objectives, Critical Use Cases, and Key Scenarios

In this section, we look at the requirements from a business point of view. A first step when executing the CloudScale method is to identify which of the system's SLOs imply the most risks from a business point of view. This is the first step of the CloudScale method. You can find it at the top of the actions in Fig. 5.2.

Risks are often functional, i.e., implementing the wrong functionality or implementing it different to customers' expectations. For example, the use case to administer the bookshop is important from a functional point of view but does not intersect with any risks concerning scalability, elasticity, or cost-efficiency. Therefore, for the CloudScale method, we need to identify use cases where those three properties are risky to implement. We call such use cases *critical*. Knowing the critical use cases is important for any kind of analysis—either model based or on the real system—because they are the major input to focus on the analysis steps. The reason is that these steps can be quite time consuming. Accordingly, it is important

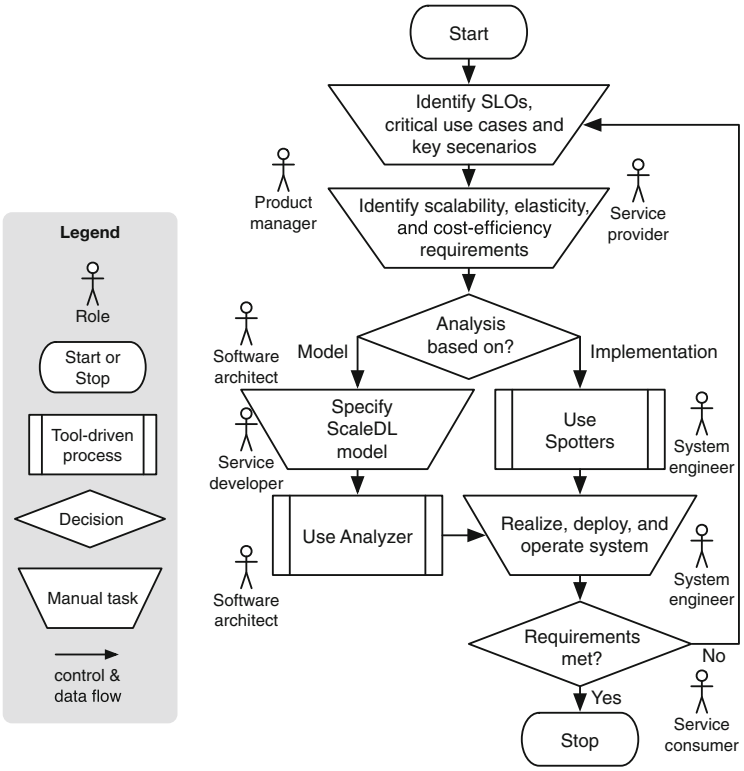

Fig. 5.2 High-level process steps and roles of the CloudScale method

to execute them not on the complete system but only on the identified critical use cases in order to keep the overall task small and manageable. As critical use cases are those which deal with high-risk scenarios, the following outlines the risks for scalability, elasticity, and cost-efficiency.

Let us focus on scalability risks first. As described in Sect. 1.3, an SLO consists of a quality metric and a quality threshold for this metric. In Sect. 2.3.1 we used the 90% response times as metric and the quality thresholds for the four CloudStore operations were between 1 and 5 s.

To identify scalability risks, software architects first need to identify rough SLOs for the most important functionality, i.e., the most important operations. Based on these SLOs, the software architect looks at these operations and finds the most critical operations, making up critical use cases. Based on rough SLOs for the four CloudStore operations in Sect. 2.3.2, as well as some reasoning, we identified the Pay operation to be most critical.

Afterward, the architect has to consider the planning horizon and based on seasonal and trend variations estimate the point in time when the work and load on the most critical operations are likely to pose the biggest threat to the scalability of the system. As this identifies a scenario for the critical use case which stresses its SLOs the most, it is called a *key scenario*. In Sect. 2.3.3, we find that in scenarios for the Pay operation, the highest load is expected to happen at noon on a Monday just before Christmas in the third year. Therefore, we assume that this will also be the key scenario used in analyses.

Once the scalability risks are known, we have to focus on elasticity risks next. Elasticity allows the system to scale according to its actual workload. Typically, risks arise in this area from insufficient adaptations or adaptation speed of the resources available to the system. Hence, we need to identify critical use cases in which rapid provisioning and deprovisioning of resources are required. For example, when a bunch of customers decide to buy books at almost the same point in time, this might need swift provisioning of additional resources and can therefore serve as a critical use case. To identify the risk, we need to figure out how long it will take for a certain rapid increase in load to increase resources as fast as possible, given their provisioning time. As the latter depends upon the capabilities of lower system layers like the IaaS layer, software architects need to identify risks that provisioning of resources in these lower layers might be slow or might happen delayed. The assessment of elasticity risks also depends on how critical it is that customers might leave our web shop, in case they do not get an answer in time due to an ongoing adaptation of the available resources. The more critical this is, the higher the architectural risk of such a scenario becomes. When specifying a key scenario for this critical use case, we need to specify how fast the load increases from the normal level to the level where many customers arrive at the bookshop in a burst.

Finally, we need to identify cost-efficiency risks of the system. These are risks where high losses of money are caused by massive over-provisioning of resources. For example, if there is a critical use case with a dramatic and rapid drop in the number of customers in a web shop but the elasticity management takes a long time

to notice this and the cost for provisioned resources is high, then this is a financial risk. Even worse, adding resources in a situation where the system faces increasing load but has already reached its capacity will cause significant financial losses, too. This setting emphasizes once more the importance of the system's capacity limits. In a key scenario of such a critical use case, you need specifications of the drop in the number of users but also information on the reaction time of the elasticity management of your system (including provisioning and deprovisioning times, the delay needed to detect the new situation, etc.).

As a result of the three identification steps, the software architect has a complete list of SLOs, associated critical use cases, and their scenarios. This is used as input in the next step to identify requirements as outlined in the following subsection.

5.7 Identify Scalability, Elasticity, and Cost-Efficiency Requirements

In Sect. 5.6 we identified business-related risky SLOs with respect to scalability, elasticity, and cost-efficiency; derived critical use cases from them; and selected key scenarios for those critical use cases. In this section, we revisit these business-related SLOs, critical use cases, and key scenarios from the previous step and enrich them with detailed workload information. As a result, we get *technical requirements* for scalability, elasticity, and cost-efficiency. In addition, we also sort these requirements according to their priority. This section, therefore, describes the second step on top of Fig. 5.2.

A requirement can be formulated on several levels of detail. For example, a generic, business-driven requirement may be that the system should be able to respond within 1 s. Here, the metric, the operations, the maximum load, as well as specifications of the work parameters are missing. However, one quality threshold is already specified. Such a requirement will therefore be open to interpretation. Nevertheless, it might make sense from a business perspective to specify them in the first place. To continue with the CloudScale method, such business-related requirements need to be revisited and detailed to reach a precise technical requirement. In this process we may differentiate between the quality thresholds among the operations. In addition, we may add information about the precise quality metric used, the operations, work parameters, and load. In practice, this will be an iterative progress between the first step and this second step in the CloudScale method.

Together with SLOs and the expected maximum workload under a specific scenario, these more detailed technical requirements guide the selection of architecture as well as implementation in the case of a new system. For an existing system, the technical requirements will guide our analysis efforts. Armed by these technical requirements, it will also be possible to test a system: is it able to manage the expected workload while adhering to the SLOs?

To identify scalability requirements, software architects need to determine the maximum workload the system should handle as constrained by the SLOs, critical

use cases, and key scenarios from the previous step. Workload covers both aspects, work and load.

There is a fundamental difference between the user point of view and the actual implementation. The requirements of a system should be formulated without considering the implementation of the system. This distinction between requirements and their realization might not be so easy in practice, because the operations constrained by SLOs can be already seen as part of the implementation. In practice, this is handled by iterations. In addition, to fully characterize the scalability, elasticity, and cost-efficiency of a service, we must also specify its configuration parameters as well as its deployments. These details are often fixed late in the development process. Therefore, feedback from these cycles might have an impact on requirements fulfillment and might require iterating them.

For dynamic properties like elasticity, it is furthermore important to know the change of the load and work over time. This is called *usage evolution* in the CloudScale method. This information needs to be captured quantitatively. In particular, for each key scenario (Sect. 5.6), the architect should capture information about the work and load evolution quantitatively.

Similarly, we also need to identify more details for work and load evolution also for the key cost-efficiency scenarios to transform them into technical requirements. Here, we are interested, in particular, in situations where the load and work decrease so that resources can be released.

After specifying the technical requirements, finally, the software architect should sort them with respect to risks for scalability, elasticity, and cost-efficiency according to priority, e.g., based on the severity of their business impact. In this way, the software architect starts with the most critical ones and first analyzes them. Note that this allocation of priorities should be done by the architecture board (i.e., architects, managers, the shop operator, and other relevant stakeholders).

When prioritizing, we also have some trade-offs concerning the granularity of the information:

Operations Some operations are much more important than other operations because of their product of work and load. Even a rare operation may be critical with a high work, and an operation with a low work may also be significant with a high load.

SLOs (service-level objectives) describing both a quality metric and a threshold for this metric. To fully specify a service, we need an SLO for all operations, but a first simplification may be to say that all operations share the same quality metric and also that they share the same quality threshold.

Load We can specify the load for all the operations individually. Another possibility is to specify the probability between the operations, together with the load on the average operation. In this case, we also need the operation mix describing the probability of each of these operations. Note that for a scalability analysis, load is often an output of the analysis.

Work The granularity of the work specification can be adjusted by putting the focus on a few work parameters only in contrast to modeling all work parameters.

For each of the identified and prioritized requirements, we run through an analysis as described in the next steps. Once the most critical use cases have been analyzed, the next critical use case might be selected from the remaining list. It then will also get analyzed. This repeats as long as critical use cases are remaining, and further analyses should be performed.

Depending on the scenario, the type of information available, in particular the availability of executable source code and usable resources, the architect makes a decision for each identified requirement: Should it be analyzed using a model of the system or should it be analyzed by inspecting the system code and executing it? Situations in which the architect wants to model a system and the steps needed in this case are detailed in Sect. 5.8. In Sect. 5.9 we look at situations in which the system's implementation is used as basis for analysis, i.e., either by source code analysis or by executing and tracing the system.

5.8 Use-Case I: Analyzing a Modeled System

One alternative to check the identified critical use cases and key scenarios for fulfillment of the risky requirements is to use a *model* of the system under study (see left branch in Fig. 5.2). There are different situations in which software architects like to gain the benefits of analyzing a system based on a model. First, in a greenfield development situation, i.e., a situation in which no preexisting implementation of the system under development exists, a model allows to analyze the critical use cases. Such a model will most likely be on an abstract level, which is often good enough for assessing the identified risky requirements. Second, a model is beneficial in cases where an implementation exists; however, the critical use case contains key scenarios, which are difficult and costly to execute, as they may require a lot of effort and resources. For example, executing a system in a high-load situation is a challenge for load generators due to the needed hardware or operating system resources. In addition, executing scenarios, in particular, elasticity or cost-efficiency scenarios, might take a long time. Third, software architects may want to use a model for quick analyses of a variety of what-if scenarios, often in order to optimize their system's behavior. Due to the large amount of slightly varied scenarios, the time it takes to analyze a running system is even multiplied. And finally, a model is also useful in cases of brownfield developments, where software architects face a combination of existing, legacy system components and non-existing, to-be-developed components.

Despite the benefits models provide in the situations outlined in the previous paragraph, there are also drawbacks of models, which might prevent their use. First, creating and parameterizing a model is a non-trivial task and requires skill and effort. The CloudScale method tries to lower these drawbacks by providing the CloudScale integrated development environment (IDE) and the CloudScale Extractor tool (details on ScaleDL extracting in Sect. 4.5.3.2), which aid in the model creation process for brownfield developments.

In case the software architect wants to use a system model, he has to execute two coarse-grained steps: First, he needs to specify the system using the ScaleDL modeling language with the aid of CloudScale's modeling tools (for details on ScaleDL modeling, refer to Chap. 4). As soon as the model is done, the software architect can use the ScaleDL model as input to the Analyzer. The Analyzer simulates whether scalability, elasticity, and cost-efficiency requirements are sufficiently achieved, thus allowing the software architect to iteratively improve the architectural model until satisfied (details in Sect. 6.3). Once satisfied, service developers and system engineers can realize, deploy, and operate the planned system with a lowered risk of violating the identified risky requirements. If system engineers—during testing or operation—still detect requirement violations, they can reiterate through the CloudScale method. Because the system has now been realized, system engineers can also decide to analyze the newly implemented system in the next iteration.

5.9 Use-Case II: Analyzing and Migrating an Implemented System

In contrast to using a model (cf. Sect. 5.8), software architects might want to analyze existing system implementations and check for the fulfillment of the identified risky requirements (see right branch in Fig. 5.2). This might be useful in the following situations. First, the system has been implemented before and faces a change in its environment; in particular, it is exposed to a critical use case which includes a usage evolution for which the system might not be prepared. In this case, the architect wants to know whether the system will still comply with its requirements under the changed load and/or work situation. Second, the system might face a planned migration. One example, which is particularly important for CloudScale, is the migration of an existing, non-cloud legacy application to a cloud computing environment. However, when migrating such a system to a cloud computing environment, the system does not automatically guarantee scalability. Scalability might be limited by a system's capacity due to existing software bottlenecks, like the ones imposed by the One-Lane Bridge HowNotTo (cf. Sect. 2.10). 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.

The benefit of using a real implementation of the system is that the analyses are representative of the studied scenario, as they do not include abstractions as models do. Therefore, the gained results are often considered more trustworthy by decision-makers. In addition, analyzing the implementation of a system often does not require as much manual effort as creating a model, in particular, in cases where the system is already installed and configured in a representative testing lab.

The drawback of analyzing the implementation of a system is that it often takes much longer until analysis results become available, as executing the system under study and collecting sufficient measurements might be time consuming and resource consuming. In addition, system engineers need to provision a representative infrastructure environment and install the application in it. Furthermore, they need to provide realistic load scripts which implement the identified key scenario under study, e.g., using Apache's JMeter.

With the CloudScale method, system engineers are able to use CloudScale's scalability detection tool called Spotter. Spotter has two subcomponents called Static Spotter and Dynamic Spotter. The Static Spotter can detect potential scalability issues at the code level without executing the code. Instead, it identifies potential scalability issues. These potential scalability issues point the Dynamic Spotter to the code it should instrument. Based on a workload generator, it 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 on which of the tests failed and for what reason they failed, the Dynamic Spotter reports a set of diagnostic statements with found scalability anti-patterns (CloudScale's HowNotTos). After software engineers eliminate detected scalability issues, they can continue to deploy the reengineered system and test for other identified critical use cases, as long as there are some left.

5.10 Realize, Deploy, and Operate

Once the selected analysis (based on Spotters or the Analyzer) indicates that scalability, elasticity, and cost-efficiency requirements are sufficiently met, the realization of the system can be completed (see lower-right corner of Fig. 5.2). This realization depends on whether a new system or an existing system is to be realized.

For new systems, system engineers realize the system based on the architectural representation in the ScaleDL model. For example, for each modeled software component, according source code has to be implemented, reused from existing components, or be externally bought. The Static Spotter may be used for static spotting of anti-patterns in the code.

For existing systems, system engineers semi-automatically reengineer detected issues based on either the Spotter or Analyzer suggestions. The Spotter points to code that requires reengineering and suggests HowTos to solve detected issues. The Analyzer suggests a revised ScaleDL model that software architects have approved to satisfy SLOs. System engineers have to then reengineer the system according to this revised model.

Once realized, the system has to be deployed and operated. The ScaleDL model particularly prescribes how a system's components are assembled and deployed to the target environment. System engineers follow this prescription and set the system in operation.

The Dynamic Spotter may be used on the system in operation to further spot anti-patterns. If new requirement violations arise, a new iteration of the method needs to be executed, again supported by the dedicated detection tools. One result of the analysis may also be a relaxation of some of the requirements and the subsequent identification of refined critical use cases and key scenarios.

If the system meets its requirements, system engineers can stop the CloudScale method and only have to reenter the method in case requirements or the system's context change.

5.11 Conclusion

This chapter introduced the CloudScale method. This method can be used to engineer cloud services, which have critical requirements with respect to scalability, elasticity, and cost-efficiency. The method supports two main usages: analyze non-existing systems on a model basis and analyze systems which have been implemented before and can be executed. The method is introduced step per step starting with requirements elicitation and ranging until system deployment and operation.

The CloudScale method provides benefits to software architects, service deployers, and service customers. It allows to check for requirements fulfillment as early as possible, thus avoiding costly rework activities. In addition, it is a useful method when migrating existing systems to the cloud. In contrast to other method, like software performance engineering (SPE), it focuses specifically on cloud computing applications. In contrast to SPE by Smith [6], for these systems, scalability, elasticity, and cost-efficiency are more important than performance. However, a good performance is the prerequisite to implement scalability, elasticity, and cost-efficiency.

In the following chapters, we outline the two use cases of the CloudScale method in more detail, give hints on how to manage introducing and executing the method, and illustrate it in case studies. In the future, the method can be extended by further steps, or it can be enriched by introducing new tools into the method's steps. For example, tools which reduce the manual effort to create and enrich ScaleDL models would have a significant impact on the method's applicability.

References

1. Brataas, G., Stav, E., Lehrig, S., Becker, S., Kopcak, G., Huljenić, D.: CloudScale: scalability management for cloud systems. In: Proceedings of International Conference on Performance Engineering (ICPE). ACM, New York (2013)
2. Brataas, G., Becker, S., Lehrig, S., Huljenić, D., Kopcak, G., Stupar, I.: The CloudScale method: a white paper (2016). http://www.cloudscale-project.eu/publications/whitepapers

3. Koziolek, H., Schlich, B., Bilich, C., Weiss, R., Becker, S., Krogmann, K., Trifu, M., Mirandola, R., Koziolek, A.: An industrial case study on quality impact prediction for evolving service-oriented software. In: Taylor, R.N., Gall, H., Medvidovic, N. (eds.) Proceeding of the 33rd International Conference on Software Engineering (ICSE 2011), Software Engineering in Practice Track. Acceptance Rate: 18% (18/100), Waikiki, Honolulu, HI, pp. 776–785 (2011). [Online] http://doi.acm.org/10.1145/1985793.1985902
4. Lilja, D.: Measuring Computer Performance. Cambridge University Press, Cambridge (2000)
5. NIST Cloud Computing Standards Roadmap. National Institute of Standards and Technology (NIST), Technical Report 500-291 (2013). http://www.nist.gov/itl/cloud/upload/NIST_SP-500-291_Version-2_2013_June18_FINAL.pdf [Visited on 06/18/2016]
6. Smith, C.U., Williams, L.G.: Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Addison-Wesley, Boston, MA (2002)