Chapter 4
ScaleDL

Gunnar Brataas, Steffen Becker, Mariano Cecowski, Vito Čuček, and Sebastian Lehrig

Abstract This chapter describes the family of languages required to analyze the scalability, elasticity, and cost-efficiency of services deployed in the cloud. First, the ScaleDL Overview Model describes the overall structure of a cloud-based architecture. Second, ScaleDL Usage Evolution specifies how load and work vary as a function of time. Third, ScaleDL Architectural Templates save time by reusing best practices. Fourth, the Extended Palladio Component Model is used for modeling software components and their mapping to underlying software services. The first three languages are new in CloudScale, while the fourth, Extended Palladio Component Model, is reused. For each language, we describe the basic concepts before we give an example. Tool support is then outlined. We list our catalog of Architectural Templates.

This chapter is structured as follows: Sect. 4.1 outlines the relation between the ScaleDL languages. For each language, we describe the basic concepts before we give an example. Tool support is also outlined. The ScaleDL Overview Model is described in Sect. 4.2. ScaleDL Usage Evolution is explained in Sect. 4.3. In Sect. 4.4 ScaleDL Architectural Templates are introduced in detail. Section 4.5 describes the Extended Palladio Component Model.

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

M. Cecowski • V. Čuček
XLAB d.o.o., Pot za Brdom 100, 1000 Ljubljana, Slovenia
e-mail: mariano.cecowski@xlab.si; vito.cucek@xlab.si

S. Lehrig
IBM Research, Technology Campus, Damastown Industrial Estate, Dublin 15, Ireland
e-mail: sebastian.lehrig@ibm.com

S. Becker et al. (eds.), *Engineering Scalable, Elastic, and Cost-Efficient Cloud Computing Applications*, DOI 10.1007/978-3-319-54286-7_4

4.1 Introduction

The Scalability Description Language (ScaleDL) is a collection of languages to characterize scalability, elasticity, and cost-efficiency aspects of cloud-based systems. For other aspects like system behavior, data models, etc., complementary languages like Unified Modeling Language (UML) must be used. ScaleDL consists of five languages: three new languages (ScaleDL Usage Evolution, ScaleDL Architectural Templates (ATs), and ScaleDL Overview Model) and two reused language (Palladio's Palladio Component Model (PCM) extended by SimuLizar's self-adaption language and Descartes Load Intensity Model (DLIM)). For each of these, we briefly describe their purpose and provide a reference to a detailed description later in this chapter:

ScaleDL Overview Model (developed in CloudScale) allows architects to model the structure of cloud-based architectures and cloud deployments at a high level of abstraction (cf., Sect. 4.2).

ScaleDL Usage Evolution (developed in CloudScale) allows service providers to specify scalability requirements, e.g., using evolution of work and load of their offered services (cf., Sect. 4.3).

Descartes Load Intensity Model (DLIM) (reused; see [1]) was originally designed to model load intensity in terms of evolution of arrival rates over time, but can also be used for modeling the evolution of work and load in general (cf. Sect. 4.3).

ScaleDL Architectural Templates (developed in CloudScale) allows architects to model systems based on best practices as well as to reuse scalability models specified by architectural template engineers (cf., Sect. 4.4).

Extended Palladio Component Model (reused; see [2]) 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 (cf., Sect. 4.5).

Figure 4.1 shows an overview of how the languages relate to each other, and the transformations and other components they are input to and output from. We will detail this in the next sections, for one language at a time.

4.2 Overview Model

Important issues while modeling cloud architectures and their deployments are their replicability and the necessity of high-level descriptions that can be easily understood and shared. Common approaches for sharing such models are diagrams and descriptions that are not useful as formal definitions of architectures or deployment strategies that can be used automatically with different tools, and formal

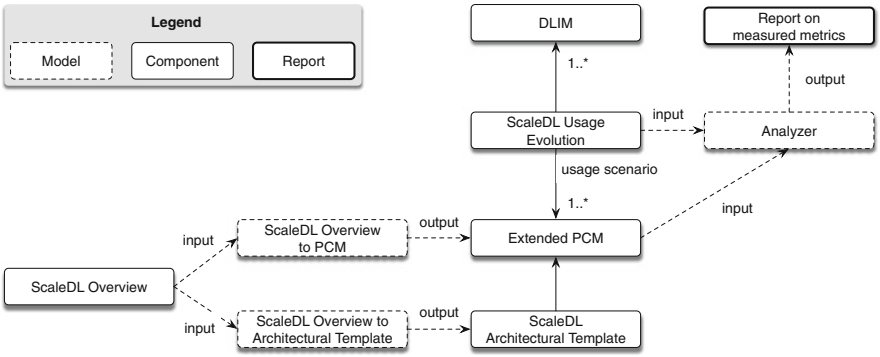

Fig. 4.1 Overview of ScaleDL languages and their relationships

descriptions, such as deployment scripts or recipes, provide little utility for the high-level study of the defined systems.

DEFINITION 4.1: OVERVIEW MODEL

The Overview model is a meta-model that provides a design-oriented modeling language and allows architects to describe the structure of cloud-based systems. It provides the possibility of representing private, public, and hybrid cloud solution, as well as systems running on a private infrastructure.

We will first describe concepts in the Overview Model in Sect. 4.2.1, before we sketch an example in Sect. 4.2.2. In Sect. 4.2.3 we detail the tool support.

4.2.1 Concepts of Overview Model

The Overview model consists of Architecture, Deployment, and Specification models. The Architecture model provides a descriptive abstraction of the system's architecture without any deployment or performance information, which is defined in Deployment and Specification models and referenced to the Architecture model.

The Architecture model was designed as a base model for describing and visualizing components in a cloud environment. It contains different cloud environments and external connections, for linking operations with a user interface or an external black-box service, or to interconnect cloud environments in a hyper-cloud configuration. The cloud environment component contains basic information about data centers and performance in different regions, using descriptors defined in the Specification model. From the architecture point of view, the most important components in the Architecture model are internal connections, defined between two components, and a layered tree structure of services, defining a deployment

hierarchy. The latter is separated into three categories: the infrastructure layer, the platform layer, and the software layer.

The infrastructure layer contains provided Infrastructure services. Services mentioned as provided in the Overview model context do not contain further information about the implementation of a lower-level mechanics. This can be substituted and described with independent components, capable of executing higher-level routines, according to measured performance limitations. A set of aforementioned components defines the Deployment. In practice, every service, except for the physical hardware, needs a lower-level service on which it operates, but sometimes the exact specification is not known. To make the Overview model flexible for such cases, the Provided service interface can be applied on any service inside the Architecture service layer stack to obscure or simplify the complexity of the underlying layers. Infrastructure services are the lowest in the service layer hierarchy, so they must provide the Deployment model. Practical implementation of the Infrastructure service is the Computing infrastructure service, which reference a Deployment and a Computing resource descriptor.

The platform and software layers contain provided or deployable platform and software services. The platform services can act as a placeholder for the software services or provide a full description of a software by describing the application's inner-working with the PCM language in Sect. 4.5.

The descriptions of cloud components inside the Architecture and the Deployment package are defined inside the Specification model to allow easy extensibility or migration and performance testing between different cloud providers.

4.2.2 Example of Overview Model

An example model can illustrate more clearly the Overview Model's capabilities. Figure 4.2 shows a visual representation of an Overview Model of a simple system composed of two Tomcat applications running on Amazon EC instances, and which make use of the Amazon DynamoDB service, a MySQL RDS service, and an e-mail service.

The model includes networking details such as average latency and bandwidth, which can be used for the behavior analysis and simulation of the overall system. Several other data can be defined. For example, we can define the expected statistical distribution of response time for an external service, or the expected capacity of a computing unit.

The general Overview Model can thus give a quick understanding of the overall architecture and deployment strategy, but contains also detailed information that can be used at different stages of the evaluation of the solution.

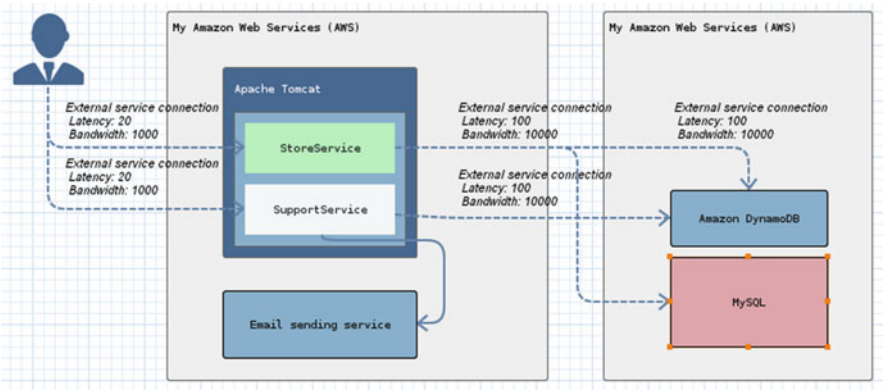

Fig. 4.2 Hybrid cloud architecture example

4.2.3 Tool Support for Overview Model

To simplify the creation and editing of the Overview model, a specially designed diagram editor with a components palette, properties view, and a number of supporting wizards has been created. The graphical diagram offers an organized view of the cloud solution architecture, and the supporting editors, together with a properties view, provides the ability to alter service descriptions.

The creation of the Overview model starts by choosing the desired cloud environment. Currently supported environments in the CloudScale Environment are Amazon web services (AWS), OpenStack, SAP Hana Cloud, and generic. The latter one contains services that are environment independent. The system architect has the ability to model hybrid cloud architectures (see Fig. 4.2) by creating multiple cloud environments in a single Overview model. When the environment is created, the architect can stack infrastructure and platform services. If the implementation of a service is described as Partial PCM model, it can simply be imported into the service component of the Architecture model. A lot of options and settings inside the properties view of the diagram are selection dependent to speed up the modeling and configuring process. More demanding, in terms of configuration, software services have dedicated editors for specifying operation interfaces, data types, and required connections.

When the Overview model is finished, it can be used for performance and cost analyses, because it contains the complete description of a cloud solution.

4.3 Usage Evolution

Existing modeling environments like Palladio [2] have usage scenarios with a fixed value for arrival rate (open) or for population (closed). Work is also fixed. If you want to analyze what happens with your service during evolution of work and load,

the current approach would be to run several simulations and manually change load and work. This manual process is time-consuming as well as error prone, as new values need to be entered in several locations of the model for each run.

Here, we propose a more direct approach using usage evolution that particularly accounts for transient phases, i.e., phases in which the system is subject to contextual changes during simulation. By *usage evolution* we mean how usage-oriented concepts like work, load, and quality thresholds vary as a function of time (Definition 4.2).

DEFINITION 4.2: USAGE EVOLUTION [3]

Usage evolution describes how usage-oriented concepts like work, load, and quality thresholds vary as a function of time.

In this section, we will first describe concepts for usage evolution in Sect. 4.3.1, then the usage evolution on an example in Sect. 4.3.2, before we describe tool support for usage evolution in Sect. 4.3.3.

4.3.1 Concepts for Usage Evolution

Figure 4.3 illustrates the meta-model for usage evolution in CloudScale. Elements imported from the PCM and DLIM are shown in light gray in the figure. The meta-model is defined to allow the specification of how the usage evolves over time. The root element of a Usage Evolution model is the *UsageEvolution* element. A *UsageEvolution* contains an ordered list of one or more *Usage* elements.

A *Usage* defines how one *UsageScenario* from a PCM model evolves over time. The referenced *UsageScenario* defines the initial values for work and load. Evolution of load for the *Usage* is described in a DLIM model (shown as a relation to the *Sequence* element from DLIM in the figure). The output values of the

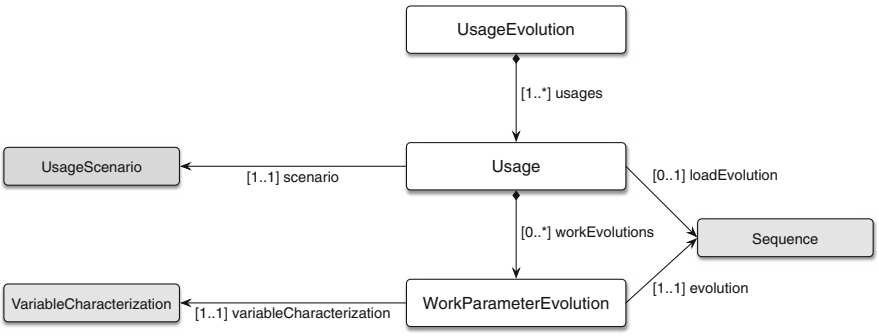

Fig. 4.3 Meta-model for Usage Evolution

DLIM model determine the evolved arrival rates in the case of open workload, and population in the case of closed workload. A *Usage* can also contain zero or more *WorkParameterEvolution* objects that each describes how a work parameter of the PCM model evolves in terms of a DLIM model. Which work parameter is to evolve is determined by a reference to a *VariableCharacterisation* defined in the PCM model.

The root element of a DLIM model is a *Sequence*, which can hold one or more function containers. Each such container holds a function for characterizing seasons and trends. Seasonal variation can be daily (peaks during lunch breaks), weekly (peaks at weekends), monthly (peaks at pay days), and yearly (peaks before Christmas). Trends describe a gradual increase or decrease and may be linear or exponential. Functions can also be combined with other functions through a list of combinators that can have addition, multiplication, or subtraction semantics. For further details about the DLIM meta-model, see [1].

4.3.2 Example of Usage Evolution

In this section, we will describe examples of usage evolution for load as well as for work. In Sect. 6.2.2 more examples will be shown.

Since CloudStore has many different operations, each of these operations could have had different load evolutions, but a natural simplification is to have one load parameter, representing the evolution of the average operation, instead of several operations evolving independently. Figure 4.4 shows how the number of users vary during a 3-min period for CloudStore. Initially, there are 2000 simultaneous users. In the first half-minute, this figure illustrates a linearly increasing trend, followed by a stable period with 5000 users for 1 min, and then a new increase up to a new stable period in the last minute. In this last stable period, there are 10,000 users.

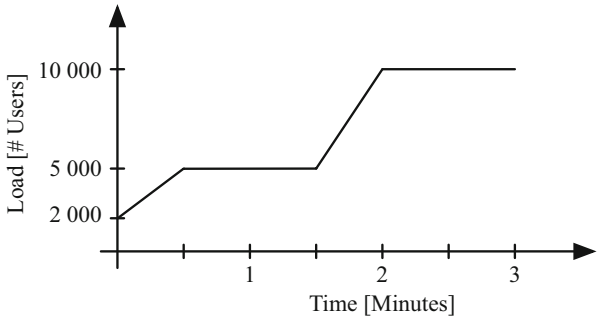

Fig. 4.4 Load evolution

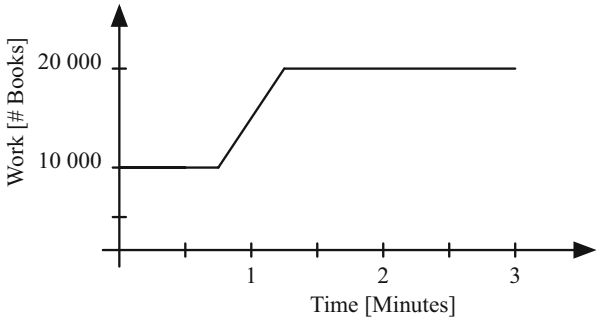

Fig. 4.5 Work evolution

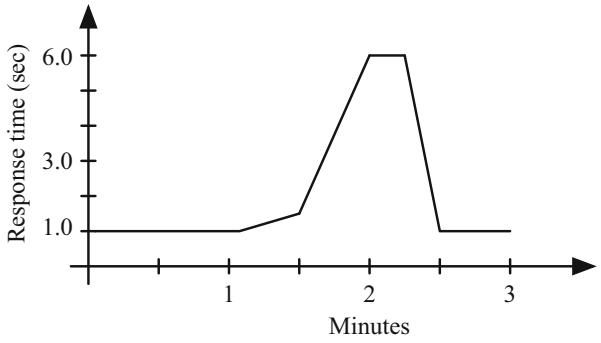

Fig. 4.6 Simulation results—response time

Figure 4.5 depicts evolution of the work parameter describing the number of books. For the first three-quarters of a minute, the number of books is stable at 10,000 books. Then, during the next half-minute, we have a linear increase up to 20,000 books, which defines the stable load during the remaining period. The reason for this sudden increase in the number of books can, for example, be the inclusion of several new publishers in the book store.

The response time for the Product Detail operation of this example is shown in Fig. 4.6 and is calculated by running the simulation based on the usage evolution model. Assume that the service-level objective (SLO) for this operation is a 90% response time of 3 s. The *x*-axis on this figure is again minutes, and the *y*-axis is the 90% response time in seconds. From the figure, we can determine that the initial increase in load is handled by the system without any increase in response time. The increase in the number of books just after 1 min results in a small increase in the response time. The further increase in the arrival rate between 1.5 and 2 min leads to a sharp increase in the response times. However, after less than 0.5 min, the response time drops again, even if neither the load nor the work on the system drops. The reason is of course that because of auto-provisioning, we now use more cloud resources. Since this auto-provisioning takes some time, we experience high

response times, before the system eventually returns to normal operation again and SLOs are no longer violated.

4.3.3 Tool Support for Usage Evolution

The load of the system is described as part of the usage scenarios in the Analyzer [2], either as a closed load, based on a fixed population and a waiting time, or as an open load described by the inter-arrival rate of new users. Work is modeled as a characterization of input and output parameters of operations, and is included in the service effect specifications (SEFFs), with some initial values defined in the usage scenario.

Palladio's usage scenarios define static values for load and work. To support evolution in terms of *variations* in load and work over time, we have extended the modeling support of Palladio by introducing a usage evolution model based on DLIM, which is used by the load intensity modeling tool LIMBO [1]. While work evolution and load evolution are explicitly modeled, other evolutions require a new simulation: change in quality thresholds, new or deprecated operations, or change in the implementation of operations. See [3] for more details.

We have added support to Palladio's simulator SimuLizar [4] such that it can run simulations following the characteristics of usage evolution models. At simulation time, SimuLizar updates workload parameters according to *Load* (as specified by *Usage* elements) and *Work Evolutions* (as specified by *WorkParameterEvolution* elements). For these updates, SimuLizar samples the linked DLIM models once per simulated time unit.

4.4 Architectural Templates

The creation of architectural models—especially with analysis capabilities—can involve huge efforts by software architects. During creation, architects may have to manually use architectural knowledge in the form of CloudScale's HowTos (cf. Sect. 2.9). Common design-time analysis approaches unfortunately lack support for directly reusing such HowTos. This lack makes the design space for software architects unnecessarily large; architects potentially consider designs that violate the constraints of HowTos. Moreover, this lack makes an automatic processing of HowTos impossible; architects have to manually model elements described in HowTos over and over again, even in recurring situations. These issues point to an unused potential to make the work of software architects more efficient.

To use this potential, the CloudScale method introduces so-called Architectural Templates (ATs) [5] for efficiently modeling and analyzing quality-of-service (QoS) properties of software architectures. With ATs, software architects can quantify

such quality properties based on reusable analysis templates (Definition 4.3) of recurring architectural knowledge such as documented in CloudScale's HowTos. Architects only have to customize such templates with parts specific to their software application, thus reducing effort and leading to a more efficient engineering approach.

DEFINITION 4.3: TEMPLATE

A template is a reusable model blueprint from which (parts of) concrete models can be instantiated.

In this section, we will first describe AT concepts in Sect. 4.4.1. An example of an AT is described in Sect. 4.4.2. Based on our catalog of HowTos in Table 2.1, we derive a catalog of ATs in Sect. 4.4.3. Tool support for ATs is outlined in Sect. 4.4.4.

4.4.1 Concepts of Architectural Templates

The AT language is a language to specify and apply templates of architectural models for model-driven design-time analyses [6]. Such templates are called Architectural Templates (ATs).

At the core, the AT language distinguishes between ATs, i.e., template types and their instances. *ATs* consist of (1) *roles* (Definition 4.4), with *parameters* and *constraints* to extend and restrict elements of architectural models; (2) a *mapping* of such roles to a semantically equivalent architectural model construct (translational semantics [7]); (3) a *documentation* that references the HowTo to be modeled, e.g., to point to the SLOs potentially impacted by the AT; and (4) an optional *default AT instance* to be used as a starting point for modeling. These constituents allow to formalize HowTos as ATs and to collect them in *AT catalogs*.

AT instances refer both to an AT and to an architectural model, e.g., a ScaleDL model, into which the AT is instantiated. AT instances particularly include a set of bindings that instantiate AT roles with bound architectural elements and actual parameters.

4.4.2 Example for Architectural Templates

Let us have a look at a concrete application of the so-called "loadbalancing" AT for component instances. The loadbalancing AT specifies a template for the loadbalancing HowTo of component instances (see Sect. 2.9). Next, we are going to apply the loadbalancing AT to our running example (CloudStore), as introduced in Sect. 2.2.

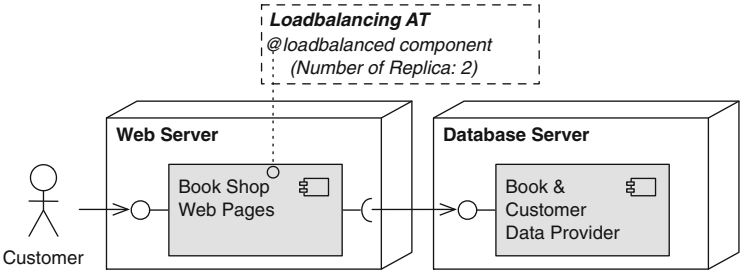

Fig. 4.7 CloudStore's architectural model annotated with the "loadbalanced component" role of the "loadbalancing" AT for component instances

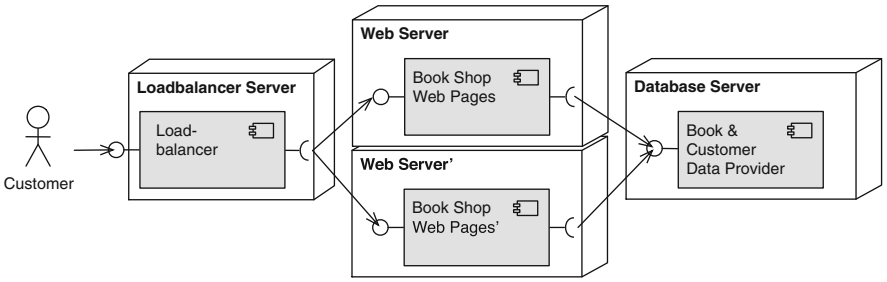

Fig. 4.8 CloudStore's architectural model after the execution of the mapping

Figure 4.7 illustrates the modified architectural model of CloudStore. In the modified version, the `Book Shop Web Pages` component instance is annotated with the `loadbalanced component` role (Definition 4.4). Moreover, we set "2" as an actual parameter value for the formal `Number of Replica` parameter.

DEFINITION 4.4: ROLE [8]

A role is the responsibility of a design element within a context of related elements.

Semantically, this model can then be mapped to the architectural model illustrated in Fig. 4.8. The new model includes a load balancer in front of the `Book Shop Web Pages` component instance. Moreover, based on the actual parameter of `Number of Replica`, the load balancer distributes workload over two copies of this component instance. The annotation of the original model is not needed anymore because its semantics have now been equivalently expressed with common elements of architectural models (such a semantic definition is called "translational semantics" [7]).

This example application of an AT shows that ATs can simplify recurring modeling tasks: instead of manually modeling a load balancer and creating two replicas, a simple, declarative annotation in the form of a role and an actual parameter is

sufficient. ATs group such recurring modeling constructs within parametrizable AT roles that can be semantically mapped back to the original constructs.

Additionally, AT roles can include constraints. These constraints allow architects to receive direct feedback during modeling. For example, whenever they annotate component instances that are stateful, the AT application may be aborted (the load balancer HowTo demands stateless component instances). Such direct feedback and restrictions reasonably limit the design space for architects, thus reducing potential modeling mistakes by architects.

4.4.3 Catalog of Architectural Templates

Based on our catalog of HowTos (see Table 2.1 in Sect. 2.9), we also derived an AT catalog with carefully engineered ATs. Table 4.1 illustrates CloudScale's AT catalog: the table provides for each AT its application domain (first column); its name, which also points to the realized HowTo (second column); and AT roles, with their parameters in parentheses (third column).

Table 4.1 CloudScale's catalog of ATs

Application domain	AT (and HowTo) name	AT Roles and parameters
Business information systems	3-Layer	Presentation layer, middle layer, Data layer
	Loadbalancing (container)	Loadbalanced container (Integer: number of replica)
	Loadbalancing (component instance)	Loadbalanced component (Integer: number of replica)
Cloud computing	SPOSAD	Presentation layer, middle layer, data layer, replicable tier
	Horizontal scaling (container)	Horizontal scaling container (Integer: number of initial replica, double: scale-in threshold, double: scale-out threshold)
	Horizontal scaling (component instance)	Horizontal scaling component (Integer: number of initial replica, double: scale-in threshold, double: scale-out threshold)
	Vertical scaling	Vertical scaling container (double: scale-up threshold, double: scale-down threshold, double: rate step size, double: minimal rate, double: maximal rate)
Big data	Hadoop MapReduce	Map component, Reduce component

The ATs of the AT catalog directly correspond to the equally named HowTos of the HowTo catalog. We therefore refer to the section about the HowTos (Sect. 2.9) for detailed descriptions of the concepts realized in these ATs. The AT roles and parameters given in the third column of Table 4.1 are directly derived from these descriptions and realize corresponding concepts.

Section 4.4.2 provides an example for the loadbalancing AT for component instances. Another and similar example is the "loadbalancing" AT for containers. This AT introduces the role of a "loadbalanced container" with a formal parameter "number of replica" of type "Integer". Software architects can accordingly attach this role to a resource container, e.g., a virtual machine. Semantically, this container is then load balanced; i.e., a load balancer is introduced that distributes workload over replicas of the container. The actual parameter that architects set for "number of replica" determines how many of these replicas exist. These semantics correspond to the according descriptions of the loadbalancing HowTo.

Analogously to these examples, the CloudScale Wiki [9] documents each AT of CloudScale's AT catalog. This documentation includes a detailed description of the related HowTo, "before mapping" and "after mapping" descriptions with according figures, and a list of concrete constraints of AT roles.

4.4.4 Tool Support for Architectural Templates

Software architects can use the graphical editors of the CloudScale integrated development environment (IDE) to apply ATs. Architects select ATs from existing AT catalogs, e.g., from CloudScale's catalog (Sect. 4.4.3). When software architects start an analysis of an architectural model with applied ATs, the mappings of ATs are automatically (and transparently to the software architect) executed.

To specify additional ATs, the CloudScale IDE provides a graphical editor to specify the elements of ATs. The mapping of an AT is specified in a separate model transformation file. As model transformation language, QVT-O [10] is currently supported.

4.5 The Extended Palladio Component Model

A unique selling point of ScaleDL is that it not only documents cloud-based systems but also allows for (semi-)automated analyses of scalability, elasticity, and cost-efficiency. ScaleDL's key ingredient for these analyses is the "Extended Palladio Component Model" (Extended PCM), an architectural description language for elastic (i.e., cloud-based) systems. Models specified with the Extended PCM can be automatically analyzed by CloudScale's Analyzer tool. The other ScaleDL parts (Overview Model, Usage Evolutions, and ATs) can be mapped to the Extended PCM to enable their analysis.

In this section, we describe concepts of the Extended Palladio Component Model in Sect. 4.5.1. Section 4.5.2 shows an example of an Extended Palladio Component Model. Tool support for the Extended Palladio Component Model is discussed in Sect. 4.5.3.

4.5.1 Concepts of the Extended Palladio Component Model

In this section, we discuss the core of the Extended PCM—the PCM itself—to understand its basic paradigms for architectural modeling. Afterward, we describe PCM extensions for elastic environments that constitute the Extended PCM.

4.5.1.1 The Palladio Component Model

The PCM [2] is an architecture description language that particularly covers performance-relevant attributes. Instances of the PCM can therefore be analyzed with respect to performance metrics like response times, utilization, and throughput.

PCM instances are constituted of partial models. Each of these partial models is inspired by the UML and covers performance-relevant attributes of the system to be modeled:

Component Specifications. Models a repository of software components. Components provide and optionally require a set of interfaces. Components can be reused whenever their provided interface is required, or exchanged whenever other components provide the same interface.

For each operation of a provided interface, components include behavior descriptions, e.g., modeling requests to operations of required interfaces, demands to resources like CPUs and hard disk drives, and acquiring and releasing connections from resource pools. These behavior descriptions are called *service effect specifications* (SEFFs).

System Model. Models a system that instantiates and assembles the software components. The system provides interfaces on its own such that users can externally access them. For implementing its provided interfaces, the system delegates requests to appropriate component instances. If these instances require further interfaces, the system includes assembly connectors that delegate requests to appropriate providing interfaces of further component instances.

Resource Environment Model. Models the resource environment (e.g., in terms of hardware) in which the system is allocated. The environment consists of containers connected via networks. Containers can, for instance, represent bare-metal or virtualized servers. Containers particularly include a set of active resources like CPUs and hard disk drives. Each of these resources comes with different processing rates and scheduling strategies.

Allocation Model. Models the allocation from component instances (system) to containers (resource environment). Therefore, the allocation specifies which container component instances demand resources.

Usage Model. Models the workload to a system in terms of its users. The usage model consists of different usage scenarios, each being either a closed workload (fixed number of users) or an open workload (users enter based on inter-arrival rates). In each usage scenario, users can access operations provided by the system. Users access such operations with a certain probability and with specific work parameters, e.g., characterizing the size of input data.

4.5.1.2 Extensions for Elastic Environments

The PCM initially was designed for static environments, i.e., for resource environments that do not change the amount of their computing resource over time. However, the usage of information systems shifted from a static to a highly dynamic behavior that challenged such static environments. For example, online shops often observe workload increases before Christmas. In such scenarios, static environments demand that resources be aligned to the maximum workload to be expected (over-provisioning). Otherwise, customers will remain unserved, which eventually leads to business losses. The disadvantage of this solution is that such an over-provisioning is expensive during non-peak times.

Cloud computing, therefore, revised the assumption that resource environments are static: to minimize expenses for resources, their amount is now elastically adapted to changing workloads. CloudScale provides PCM extensions for modeling and analyzing these elastic resource environments. CloudScale's modeling extensions cover workloads that change over time (dynamic usage environments), self-adaptation rules that react on these changes by adapting the amount of resources, and monitors to trigger self-adaptation rules:

Usage Evolution Model. Usage Evolutions specify how workload parameters of PCM usage models change over time. For example, steadily increasing and periodically varying arrivals of users can be modeled. Section 4.3 details and exemplifies Usage Evolutions.

Self-Adaptation Rules. Self-adaptation rules react on changes of the monitored usage or resource environment. For example, when a certain response time threshold is exceeded, a self-adaptation rule could trigger a scaling-out of bottleneck components. These rules, therefore, consist of two parts, a trigger and an action that can be activated by the trigger. The trigger relates monitored values to pre-specified thresholds to determine whether to activate the action. The action describes the change in the system to be applied. Actions are formulated in terms of model-to-model transformation languages like QVT-O [10], StoryDiagram [11], and Henshin [12].

Monitors. Monitors describe which metrics should be recorded at specific measuring points. Monitors can, for instance, measure metrics like utilization of a specific CPU. The resulting measurements are used as input to the trigger of self-adaptation rules, which then potentially activates an adaptation action.

4.5.2 Example for the Extended Palladio Component Model

In this section, we describe the elements of the Extended PCM and Architectural Templates used by a model of CloudStore. Figure 4.9 gives a simplified high-level overview of these elements.

With this overview, software architects can easily follow the control and data flow (arrows) from customers through CloudStore's components (UML component symbols) allocated on various resource containers (UML node symbols). In Fig. 4.9, customers enter CloudStore via the `Book Shop Web Pages` component to browse and order books. To provide its functionality, `Book Shop Web Pages` requests information from the `Book Shop Business Rules` component. `Book Shop Business Rules` can in turn request payment services from an externally hosted `Payment Gateway`. Additionally, it can request data about books and customers from the `Book & Customer Data Provider` component. If a web page returned by `Book Shop Web Pages` references images, a customer's browser subsequently fetches these references via the `Book Image`

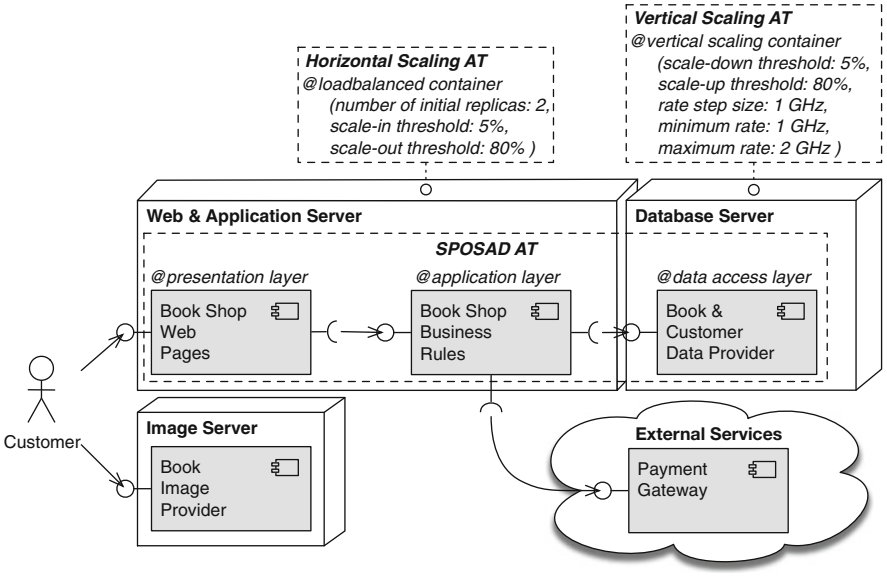

Fig. 4.9 Simplified ScaleDL model of the CloudStore online bookshop with annotated AT roles

`Provider` component. As illustrated in Fig. 4.9, `Book Shop Web Pages` and `Business Rules` are allocated on a `Web & Application Server`, `Book & Customer Data Provider` on a `Database Server`, and `Book Image Provider` on a dedicated `Image Server`.

All of these elements come from the Extended PCM: customers entering the system (Usage Model), components (Component Specifications) instantiated and assembled to CloudStore (System Model), and the allocation of these instances (Allocation Model) to different resource containers (Resource Model). As software architects, we are interested in analyzing the impact on CloudStore's QoS properties when applying architectural knowledge. Therefore, Fig. 4.9 additionally illustrates elements from ScaleDL's AT language: applications of the ATs *SPOSAD*, *Horizontal Scaling*, and *Vertical Scaling* are annotated (bold italic text in dashed boxes).

The *SPOSAD* AT (middle of Fig. 4.9) introduces roles to structure CloudStore into a *presentation layer* (bound to `Book Shop Web Pages`), an *application layer* (bound to `Book Shop Business Rules`), and a *data access layer* (bound to `Book & Customer Data Provider`). These roles constrain the bound components to only access the respective lower-level layer (in Fig. 4.9 shown from left to right). Moreover, the *SPOSAD* AT requires components on the presentation and application layers to be stateless. Because ATs formalize such constraints, an architecture tool with AT support (like the CloudScale IDE; cf. Sect. 8.7) can ensure their fulfillment, e.g., by forbidding direct connections from `Book Shop Web Pages` to `Book & Customer Data Provider`.

The *Horizontal Scaling* AT (top middle of Fig. 4.9) introduces a *loadbalanced container* role bound to the `Web & Application Server`. In a preprocessing step of a design-time analysis, a template engine will reflect the performance impact of this binding by creating a load balancer in front of the container that distributes workload. According to the parameters given in Fig. 4.9, the load balancer initially distributes workload over two container replicas and dynamically decreased or increased this number if the CPU utilization of the container drops below 5% or exceeds 80%, respectively.

The *Vertical Scaling* AT (top right of Fig. 4.9) introduces a *vertical scaling container* bound to the `Database Server`. A template engine will create adaptation rules that dynamically increase or decrease the processing rate of this container's CPU. The rate is decreased if CPU utilization drops below 5% and increased if it exceeds 80% (see the role's parameters in Fig. 4.9). These adaptations come in steps of 1 GHz within a range of 1–2 GHz (hence, the rate is either 1 or 2 GHz).

4.5.3 *Tool Support for the Extended Palladio Component Model*

Software architects can specify instances of the Extended PCM with the graphical editors of the CloudScale IDE. Once specified, architects can use various analysis tools to inspect the QoS properties of a modeled system. Moreover, if the source code of a system is already available, CloudStore's Extractor can be used to automatically create partial instances of the Extended PCM. This section briefly describes the analysis tools and the Extractor for the Extended PCM.

4.5.3.1 Analysis Tools

PCM instances serve as input (Fig. 4.10) (left) to various analysis tools (Fig. 4.10) (right):

Analyzer. CloudScale's Analyzer is a simulation of the modeled system. The simulation interprets the input PCM instance to provide measurements for performance metrics like response times. Because the Analyzer interprets PCM instances, it can also acknowledge changes of these instances during simulation time. This feature, therefore, allows to model self-adaptive systems: the execution of a self-adaption action transforms a current PCM instance into an adapted version. The Analyzer subsequently continues by simulating the adapted version. Moreover, the Analyzer supports ScaleDL's Usage Evolution models: at simulation time, the Analyzer updates workload parameters according to an input Usage Evolution model. For these updates, the Analyzer samples the Usage Evolution

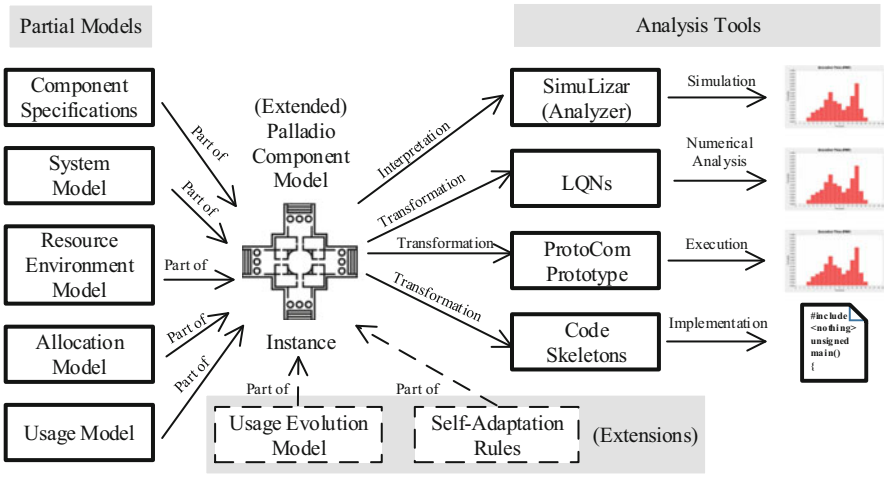

Fig. 4.10 Instances of the Extended PCM serve as input to various analysis tools

model once per simulated time unit to receive the concrete workload parameter for the current simulation time.

LQNs. Layered queuing networks (LQNs) extend queuing networks with layered structures and related elements, e.g., to fork/join different layers. Based on input PCM instances, transformations can create LQN models. These models can then be solved with numerical mean-value approximation methods, e.g., to provide mean response times as output. In contrast to simulations, these methods require less time for analyses; however, they provide only information about mean values.

ProtoCom Prototype. ProtoCom transforms PCM instances into runnable performance prototypes. Such performance prototypes can execute in different target environments and mimic demands to different types of hardware resources. Their execution, therefore, allows to take performance measurements for an early assessment of the modeled software system within a real environment.

Code Skeletons. Based on a PCM instance, a transformation generates appropriate code skeletons. These skeletons serve developers as a starting point for implementing the modeled system. Code skeletons are therefore especially important in forward engineering (see Chap. 6).

4.5.3.2 Extractor

ScaleDL models can be created manually. This is described in more detail in Chap. 6. However, CloudScale's Extractor tool can assist in potentially tedious manual tasks, given that source code is available.

The Extractor is a reverse engineering tool for the automatic extraction of partial Extended PCM models, thus lowering modeling effort for system engineers if source code already exists. The Extractor is based on the Archimetrix approach [13] that combines different reverse engineering approaches to iteratively recover and reengineer component-based software architectures.

The inputs to the Extractor are source code and configuration parameters for reverse engineering, e.g., thresholds that specify when to cluster classes into components. Software architects particularly have to decide which part of the system should be extracted. In a large system, architects may only be interested in a few critical services.

Once configured, software architect can start the Extractor. After parsing the source code, the 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.

While the Extractor relieves software architects from potentially tedious modeling tasks, software architects need to spend some effort finding the right configuration parameters.

In general, software architects start with the Extractor's default configuration and assess whether the resulting Extended PCM model is satisfying for their system. Software architects are typically unsatisfied if the result is too abstract (e.g., the Extractor clustered the whole system into one component) or too fine-grained (e.g., the Extractor clustered each class into a dedicated component). In that case, software architects alter configuration parameters and rerun the Extractor until satisfied.

The main parameters for the Extractor are (default values included):

Clustering Merge Threshold Max (End Value) (100) Start threshold between 0 and 100 for deciding whether to merge the elements of a component candidate into a single component by melting the component candidates in a single component. The lower the value is the fewer components are merged into single components.

Clustering Merge Threshold Min (Start Value) (45) End threshold between 0 and 100 for deciding whether to merge the elements of a component candidate into a single component. The lower this value is the more likely less connected component candidates will be merged into a single component.

Clustering Merge Threshold Increment (10) The increment between 0 and 100 for the merging components. The Extractor will merge components using a threshold starting at the start value and ending at the end value using this increment.

Clustering Composition Threshold Max (Start Value) (100) The start threshold between 0 and 100 for deciding whether to compose the elements of a component candidate into a new composed component by linking the component candidates using connectors. The lower the value is the fewer components are composed into a new composed component.

Clustering Composition Threshold Min (End Value) (25) The end threshold between 0 and 100 for deciding whether to compose the elements of a component candidate into a composed component. The lower this value is the more likely less connected component candidates will be composed into a composed component.

Clustering Composition Threshold Decrement (10) The increment between 0 and 100 for the composing components. The Extractor will compose components using a threshold starting at the start value and ending at the end value using this increment.

The Extractor has additional parameters which characterize the coupling of component candidates which need to be adjusted for each project to be extracted. A description of these parameters can be found in the CloudScale user manual [14].

4.6 Conclusion

This chapter presents ScaleDL as of a family of related languages. The ScaleDL Overview Model describes the overall structure of a cloud-based architecture. ScaleDL Usage Evolution specifies how load and work vary as a function of time. ScaleDL ATs save modeling efforts by reusing formally captured best practices. The Extended Palladio Component Model is used for modeling software components and their mapping to underlying elastic software services.

With the ScaleDL family of languages, software architects can specify critical aspects of a software system to enable analysis of scalability, elasticity, and cost-efficiency. Software architects may model the complete software system using all the languages, but selectively using a subset (or only fragments) of languages is also possible.

Subsequent chapters show how ScaleDL is integrated in the CloudScale method and how CloudScale's tools utilize ScaleDL. In particular, Chap. 8 describes how ScaleDL may be extended in the future.

References

1. Kistowski, J.V., Herbst, N., Zoller, D., Kounev, S., Hotho, A.: Modeling and extracting load intensity profiles. In: Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS '15, pp. 109–119. IEEE, New York (2015)
2. Becker, S., Busch, A., Brosig, F., Burger, E., Durdik, Z., Heger, C., Happe, J., Happe, L., Heinrich, R., Henss, J., Huber, N., Hummel, O., Klatt, B., Koziolek, A., Koziolek, H., Kramer, M., Krogmann, K., Küster, M., Langhammer, M., Lehrig, S., Merkle, P., Meyerer, F., Noorshams, Q., Reussner, R.H., Rostami, K., Spinner, S., Stier, C., Strittmatter, M., Wert, A.: In: Reussner R.H., Becker, S., Happe, J., Heinrich, R., Koziolek, A., Koziolek, H., Kramer, M., Krogmann, K. (eds.) Modeling and Simulating Software Architectures – The Palladio Approach, 408 pp. MIT, Cambridge, MA (2016). [Online] http://mitpress.mit.edu/books/modeling-and-simulating-software-architectures
3. Brataas, G., Stav, E., Lehrig, S.: Analysing evolution of work and load. In: QoSA: Conference on the Quality of Software Architectures. ACM, New York (2016)
4. Becker, M., Becker, S., Meyer, J., SimuLizar: design-time modelling and performance analysis of self-adaptive systems. In: Proceedings of Software Engineering 2013 (SE2013), Aachen (2013)
5. Lehrig, S.: Architectural templates: engineering scalable saas applications based on architectural styles. In: Gogolla, M. (ed.) Proceedings of the MODELS 2013 Doctoral Symposium Co-located with the 16th International ACM/IEEE Conference on Model Driven Engineering Languages and Systems (MODELS 2013), Miami, October 1, 2013. CEUR Workshop Proceedings, vol. 1071, pp. 48–55 (2013). CEUR-WS.org [Online]. http://ceur-ws.org/Vol-1071/lehrig.pdf
6. Lehrig, S.: Applying architectural templates for design-time scalability and elasticity analyses of saas applications. In: Proceedings of the 2Nd International Workshop on Hot Topics in Cloud Service Scalability. HotTopiCS '14, Dublin, pp. 2:1–2:8. ACM, New York (2014). [Online] http://doi.acm.org/10.1145/2649563.2649573

7. Kleppe, A.: Software Language Engineering: Creating Domain-Specific Languages Using Metamodels, 1st edn. Addison-Wesley, Upper Saddle River, NJ (2008)
8. Buschmann, F., Henney, K., Schmidt, D.: Pattern-Oriented Software Architecture: On Patterns and Pattern Languages. Wiley, New York (2007)
9. CloudScale Wiki: HowTos. http://wiki.cloudscale-project.eu/HowTos [Visited on 12/19/2016]
10. Object Management Group (OMG), Meta Object Facility (MOF) 2.0 — Query/View/Transformation Specification. Technical Report OMG Document Number: formal/2011-01-01 (2011). http://www.omg.org/spec/QVT/-1.1/
11. von Detten, M., Heinzemann, C., Platenius, M., Rieke, J., Travkin, D., Hildebrandt, S.: Story diagrams - syntax and semantics, Software Engineering Group, Technical Report, Heinz Nixdorf Institute, University of Paderborn (2012)
12. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: advanced concepts and tools for in-place emf model transformations. In: Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems: Part I. MODELS'10, Oslo, pp. 121–135. Springer, Berlin (2010). [Online] http://dl.acm.org/citation.cfm?id=1926458.1926471
13. Platenius, M.C., Von Detten, M., Becker, S.: Archimetrix: improved software architecture recovery in the presence of design deficiencies. In: Software Maintenance and Reengineering (CSMR 2012), pp. 255–264. IEEE, New York (2012)
14. CloudScale: User Manual of the CloudScale Environment. http://www.cloudscale-project.eu/media/filer_public/2016/02/01/cloudscaleenvironment-userguide_1_1.pdf [Visited on 12/19/2016]