

## Chapter 2

# CloudScale Method Quick View

**Gunnar Brataas, Steffen Becker, Mariano Cecowski, Darko Huljenić, Sebastian Lehrig, and Ivana Stupar**

**Abstract** In this chapter, we overview the complete CloudScale method and show how the CloudScale method relates to existing development processes. Our overview is accompanied by a running example termed CloudStore—a simple online bookshop to be operated in a cloud computing environment. In a fictional scenario, we exemplify how a software architect follows the CloudScale method to realize CloudStore. The architect finally realizes CloudStore such that all of its scalability, elasticity, and cost-efficiency requirements are fulfilled. After having exemplified the CloudScale method, additional guidelines for software architects are given in the form of best practices (HowTos) and common pitfalls (HowNotTos). The chapter closes with a discussion on how the CloudScale method can be integrated into existing development processes such as the Unified Process.

This chapter is structured as follows. Section 2.1 overviews the process steps of the CloudScale method, and Sect. 2.2 introduces CloudStore as a running example. Afterward, the fictional scenario starts in which a software architect identifies critical use cases and key scenarios (Sect. 2.3), derives appropriate service-level objectives (SLOs) (Sect. 2.4), creates an architectural model of CloudStore

---

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

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

M. Cecowski  
XLAB d.o.o., Pot za Brdom 100, 1000 Ljubljana, Slovenia  
e-mail: [mariano.cecowski@xlab.si](mailto:mariano.cecowski@xlab.si)

D. Huljenić • I. Stupar  
Ericsson Nikola Tesla, Krapinska 45, 10000 Zagreb, Croatia  
e-mail: [darko.huljenic@ericsson.com](mailto:darko.huljenic@ericsson.com); [ivana.stupar@ericsson.com](mailto:ivana.stupar@ericsson.com)

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

(Sect. 2.5), improves this model based on analyses (Sect. 2.6), implements CloudStore and resolves implementation issues (Sect. 2.7), and deploys and operates CloudStore in a cloud computing environment (Sect. 2.8). Afterward, Sect. 2.9 describes HowTos, and Sect. 2.10, HowNotTos. Finally, the CloudScale method is related to the Unified Process in Sect. 2.11.

## 2.1 Process Steps of the CloudScale Method

After briefly motivating the CloudScale method in Sect. 1.10, this section gives a high-level overview of the CloudScale method based on Fig. 2.1.<sup>1</sup> Figure 2.1 illustrates the control and data flow (denoted as arrows) for software architects who want to follow the CloudScale method. Various nodes denote a flow's start/end (rounded rectangles), processes supported by tools (rectangles with double-struck vertical lines), decisions for software architects (diamonds), and manual tasks (trapezoids).

The CloudScale method deals with scalability, elasticity, and cost-efficiency of a system and must therefore be embedded in a wider method where the functionality of the system is specified. The functionality of a service is expressed as one or more operations offered by the service. Therefore, the first step of Fig. 2.1 is to look at these operations and identify the most important operations from a performance point of view—these are the critical use cases. As a prerequisite for finding the most critical operations, software architects have to establish rough service-level objectives (SLOs) for these operations. Later, software architects must also estimate where, in the planning horizon, these critical use cases are most likely to be toughest—these are the key scenarios.

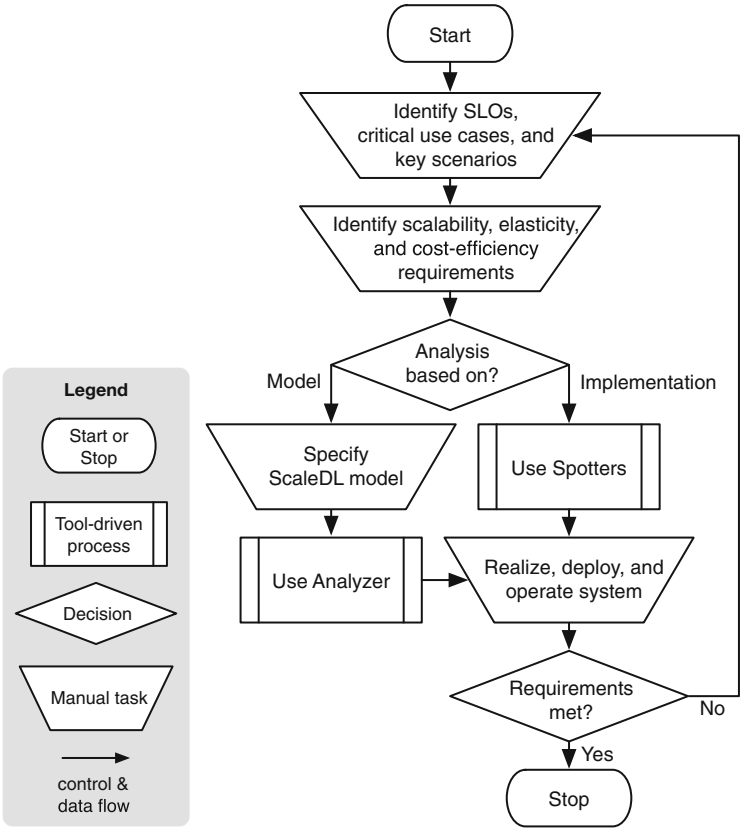
In the second step of Fig. 2.1, critical use cases and key scenarios allow software architects to define coarse requirements for scalability, elasticity, and cost-efficiency. Software architects formulate their requirements via SLOs (as described in Sect. 1.3).

The first decision node of Fig. 2.1 branches based on the type of artifacts to be studied. If the analysis shall be based on modeling, software architects go to the left, and if the analysis shall be based on implementation artifacts, software architects go to the right.

A model in the CloudScale method is an architectural model expressed in the Scalability Description Language (ScaleDL)—software architects have to specify such a model when following the left path in Fig. 2.1. ScaleDL is comparable to a Unified Modeling Language (UML) model with dedicated annotations for quality analyses, similar to UML's MARTE profile [4]. ScaleDL represents the structure and behavior of the system's architecture, the workload by the users, as

---

<sup>1</sup>The CloudScale method extends the Q-ImPRESS method [1] and builds on the Palladio performance modeling tool [2]. A first draft of the CloudScale method was introduced in [3].



**Fig. 2.1** High-level process steps of the CloudScale method

well as hardware and software resource demands. In addition, ScaleDL specifies how workload changes over time and how autonomous elasticity managers behave. ScaleDL is described in more detail in Chap. 4.

The ScaleDL model can either be a complete new design or a refinement of an earlier architectural model. Moreover, a model can be partially extracted from existing implementations via the so-called Extractor tool for reverse engineering. Partially extracted ScaleDL models must be completed manually inside the “Specify ScaleDL model” process step.

Modeled systems are analyzed in the “Use Analyzer” process step in Fig. 2.1. This analysis may reveal a perfect system, or one or more weaknesses. When these weaknesses have been corrected on the model level, the model can be analyzed again. Moreover, analyses may reveal that some of the requirements are hard to fulfill. Based on negotiations with the relevant stakeholders, alleviated requirements may then be derived and analyzed. In this way, important trade-offs between cost

and quality can be resolved before the system is put in operation where users potentially react furiously as a result of unsatisfying user experience.

Alternatively, if the analysis shall be based on implementation, software architects continue with the “Use Spotters” process step of Fig. 2.1, where anti-patterns are detected and handled using the so-called Spotter tool. The Spotter tool has two parts: the Static Spotter examines static code, while the Dynamic Spotter comprises instrumentation and load generation of a running service. In both cases, software architects reengineer the code for spotted anti-patterns by fixing their root causes.

The second decision node of Fig. 2.1 checks (based on analysis result of Spotter or Analyzer) whether scalability, elasticity, and cost-efficiency requirements can be sufficiently met. If met, the realization of the system can be completed and the CloudScale method stops. For new systems, software architects realize the service based on the architectural representation in the ScaleDL model. After realizing a service, the Static Spotter may be used for static spotting of anti-patterns in the code. For existing systems, software architects semi-automatically reengineer detected issues based on either Spotter or Analyzer suggestions. A realized system will be deployed. After deployment, you may use Dynamic Spotter on the deployed service, which may spot further anti-patterns.

The CloudScale Method has one feedback loop. If, during operations, new requirement violations arise, a new iteration of the method needs to be executed, again supported by dedicated detection tools. A 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, software architects can stop the CloudScale method, and only have to reenter the method in case requirements or the system’s environment change.

In later chapters, we refine some of the core steps in the method. These refinements will further introduce feedback loops.

## 2.2 Running Example

This section introduces a running example, which will—later in this book—be reused, revisited, and extended. The running example is termed CloudStore—an online bookshop to be deployed in a cloud computing environment. In that sense, one can think of CloudStore as a simplified variant of Amazon.

CloudStore’s book-selling services have four core operations:

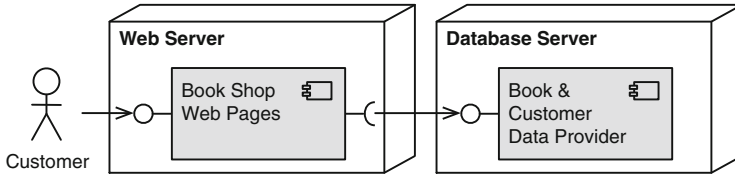
**Home Page** Provide the home page of CloudStore.

**Search** Look for a suitable book.

**Shopping Cart** Put a book in the shopping cart.

**Pay** Check-out the books in the shopping cart and get them shipped.

These core operations represent the basic functionality of CloudStore. Other operations for registering customers, order inquiry, etc. are also required but will not be considered now.



**Fig. 2.2** Conceptual CloudStore architecture

CloudStore has two work parameters: number of customers and number of books. With more customers and more books, each operation in CloudStore will be heavier.

The overall structure of CloudStore is depicted in Fig. 2.2. A customer makes use of the CloudStore service via a web browser and most of the functionality is handled by a Web Server. This Web Server connects to a Database Server storing information about customers and the book items of the shop. For now, it is assumed that payment is handled internally by the Web Server.

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

A software architect plans to realize CloudStore as introduced in Sect. 2.2. This section outlines how the software architect can start following the CloudScale method by identifying CloudStore’s business-related requirements.

The software architect first establishes the SLOs and, afterward, the critical use cases or operations from a scalability, elasticity, and cost-efficiency point of view. The software architect also identifies the key scenarios where the load, as well as work, on this critical operation becomes highest.

### 2.3.1 Service-Level Objectives

As a first step, the software architect has to estimate rough SLOs for the four CloudStore operations introduced in Sect. 2.2. As described in Sect. 1.3, an SLO consists of a quality metric and a quality threshold for this metric. The software architect selects 90 percentile response times as a suitable metric. As a first approximation, the software architect wants the Home Page and the Shopping Cart operations to respond in 1 s for 90% of requests to these operations. The Search operation has to complete in 2 s, and the Pay operation has to complete in 5 s.

### 2.3.2 *Critical Use Cases*

Critical use cases correspond to the critical operations in CloudStore. The software architect inspects the four CloudStore operations described in Sect. 2.2 and assesses their risks:

**Home Page** Providing the home page will essentially be a read operation and does not require any writes to the database. Moreover, this read operation can easily be cached, and it is also not critical if the content is not completely up to date.

**Search** Searching for a suitable book is also a read operation, but the stock level must reflect the actual situation. Realizing this requirement should be trivial in a well-indexed database.

**Shopping Cart** Putting a book in the shopping cart involves some writing to the database.

**Pay** To pay for books in a secure way is clearly the most complex operation. This is a transaction that involves several different payment systems. Moreover, the Pay operation is mission critical. If the Pay operation goes wrong, involved stakeholders potentially lose money. Customers also get very upset if there are problems with payments, much more than with the other functionality of a website. The Pay operation thus has to work.

When considering these four operations, the Pay operation is clearly the most challenging. Even if the SLO for this operation, as described in Sect. 2.3.2, is most flexible, the software architect still considers this operation to be the most challenging. To reduce the complexity in later analyses, the software architect focuses on this operation.

### 2.3.3 *Key Scenarios*

To establish key scenarios for scalability, the software architect first establishes the length of the planning horizon. The software architect uses a 3-year planning horizon for CloudStore's services.

Subsequently, the software architect must find the most critical point in the planning horizon. The software architect inspects when in the 3-year planning horizon CloudStore's requirements become toughest. As already established in Sect. 1.3, work, load, and quality thresholds for CloudStore's quality metrics are the key ways of characterizing a service in terms of scalability, elasticity, and cost-efficiency.

When it comes to load, the load of an operation often has seasonal variations. These variations typically span several time scales. The software architect identifies where in these seasonal variations CloudStore will have the highest load:

**Yearly variation** Most customers approach CloudStore just before Christmas.

**Monthly variation** Most customers approach CloudStore at salary, social security, and pension days, which is often the 20th of every month.

**Weekly variation** Most books are bought on Mondays. Later in the week, the sale gradually drops until it reaches a new peak on Mondays.

**Daily variation** Requests peak at noon, i.e., during lunch break.

In addition, the software architect identifies an exponential, yearly increase trend in the 3-year planning horizon. Therefore, based on the seasonal variation as well as the trend, the highest load on the Pay operation is expected to happen at noon on a Monday just before Christmas in the third year. The software architect assumes that this situation will also be a critical scenario for work and regarding CloudStore's SLOs.

Elasticity concerns the ability of CloudStore's services to handle sudden increases in workload while still fulfilling CloudStore's SLOs. When establishing key scenarios for elasticity, the software architect therefore considers sudden changes in load and work. The software architect first looks at sudden changes in load. The arrival of a new best-seller book typically leads to a peak in load.

When it comes to the two work parameters described in Sect. 2.2, the number of customers is connected to the number of payment operations. The software architect consequently inspects key scenarios regarding the number of books. The number of books can suddenly grow when customers buy many new books at once. Realistically, the software architect expects this to happen when CloudStore buys books from a new publisher.

SLOs must also be handled at an acceptable operational cost. The software architect therefore considers key scenarios for cost-efficiency. To be on the safe side, the software architect could use the same large deployment configuration throughout the day. However, CloudStore will not take advantage of scaling down during the night or at other times when CloudStore has less than peak load. In our example, the cost-efficiency requirements will be as follows: we take the cost required for handling the maximum daily load and divide it by 2. On average, the cost shall only be half as much as the cost required to satisfy the maximum daily load. With this cost-efficiency requirement, the simple option of using the same configuration at all times is no longer feasible.

## 2.4 Identify Scalability, Elasticity, and Cost-Efficiency Requirements

The software architect next derives technical requirements from the business-related requirements of Sect. 2.3. Of course, the software architect must ensure that the functionality and non-functional requirements like security and maintainability are fulfilled. However, motivated by the focus of this book, the software architect focuses on scalability, elasticity, and cost-efficiency requirements.

As described in Sect. 1.3, load and work are key service properties from the point of view of scalability, elasticity, and cost-efficiency. In this section, the software architect therefore determines the expected load and work for CloudStore.

### 2.4.1 *Scalability Requirements*

In Sect. 2.3.3, the software architect has decided that the planning horizon is 3 years long. For this planning horizon, the software architect estimates the largest load and work and the toughest quality thresholds for CloudStore’s quality metrics. The software architect’s estimates then allow to specify scalability requirements.

The software architect has already identified that

- the Pay operation is the most complex (cf. Sect. 2.3.2),
- the 90 percentile metric with a threshold of 5 s is used for the Pay operation (cf. Sect. 2.3.1), and
- the highest load on the Pay operation is expected to happen at noon on a Monday just before Christmas in the third year (cf. Sect. 2.3.3).

Based on experience and available data, the software architect next estimates the maximum load at this particular time. Historical data and the software architect’s best business-forecasting tools indicate that the load will be 1000 simultaneous customers for the Pay operation.

CloudStore has two *work parameters*: number of books and number of customers. The software architect expects the total number of customers to be proportional to the load. Based on experience, this number is 1000 times the number of simultaneous Pay operations, which gives 1,000,000 customers in total. When it comes to the work parameter “number of books”, the software architect simply expects a gradual growth. This means that CloudStore has the highest number of books at the end of the planning horizon. Here, the software architect expects 1,000,000 books indexed in CloudStore’s database.

In summary, the software architect’s scalability requirement can be formulated as the ability to handle 1000 simultaneous users on the Pay operation with 1,000,000 customers and 1,000,000 books. Here, handling refers to CloudStore’s SLO—i.e., Pay fulfills the less-than-5 s 90 percentile response time SLO. The software architect will later investigate this scalability requirement via modeling.

### 2.4.2 *Elasticity Requirements*

Elasticity concerns the ability of CloudStore’s services to handle sudden increases in workload while still fulfilling CloudStore’s SLOs. The software architect first investigates elasticity requirements for sudden changes in load.



The arrival of a new best-seller book typically leads to a peak in load (cf. Sect. 2.3.3). During such a scenario, the software architect expects that CloudStore must handle 100% more payment operations compared with what it must normally handle. In Sect. 2.4.1, the software architect identified a maximum of 1000 simultaneous customers on the `Pay` operation. Based on this maximum, CloudStore clearly must handle a peak of 1000 simultaneous customers on the `Pay` operation. However, the software architect also needs some estimation of how fast this peak will build up. The software architect guesses that, during a period of 1 min, the number of simultaneous `Pay` operations can grow from 500 to 1000. The software architect therefore formulates the elasticity requirement that CloudStore must be able to handle such load variations.

When it comes to the work parameter “number of customers”, this requirement is connected to the number of payment operations by the factor 100. That is, CloudStore must be able to go from 500,000 customers to 1,000,000 customers within 1 min.

The number of books can suddenly grow when CloudStore provisions many new books at once. As outlined in Sect. 2.3.3, the software architect expects such a situation when CloudStore starts to buy books from a new publisher. CloudStore will then provision 100,000 new books at once. The software architect particularly expects that this number of books will be added during 1 week. Compared with the increase in terms of both load and customers, the increase in the number of books is so slow that the software architect can simply ignore this increase when considering elasticity requirements.

In summary, the software architect formulates the elasticity requirement that CloudStore must be able to handle both an increase from 500 to 1000 simultaneous `Buy` operations and an increase from 500,000 customers to 1,000,000 customers within 1 min.

### 2.4.3 *Cost-Efficiency Requirements*

Cost-efficiency covers the cost for handling CloudStore’s workload while fulfilling its SLOs. In Sect. 2.3.3, the software architect formulated the cost-efficiency requirement to pay only half the cost required for handling the maximum daily load (assuming we used the same configuration throughout the day). This requirement is detailed enough from a technical point of view, so the software architect adds no new details here.

The software architect will next inspect the scalability, elasticity, and cost-efficiency requirements, given the expected load, work, and SLOs.

## 2.5 Specify ScaleDL Model

The software architect next checks whether CloudStore can handle the scalability, elasticity, and cost-efficiency requirements as formulated in Sect. 2.4. The software architect may use benchmarking by testing an implemented CloudStore that uses a wide selection of cloud computing resources. In the software architect's case, CloudStore is not fully developed. Therefore, the software architect decides to base the analysis of SLOs on a ScaleDL model.

The ScaleDL model basically contains the following information:

**Usage Evolution** It describes the anticipated usage of CloudStore. It has already been described at an overall level in Sect. 2.3.3, but in this step, the software architect will go deeper into the expected evolution of work, load, and potentially also SLOs.

**Component Model** It describes the relations between the software components inside of CloudStore.

**Resource Model** It describes the hardware and software resources in the underlying cloud resources for the Web Server and the Database Server, as well as how the components relate to these hardware resources. It must reflect the properties of the specific cloud computing resources used for both the Web Server and the Database Server.

This understanding of the ScaleDL model is extended in Chap. 4.

As described in Sect. 2.1, software architects can create a ScaleDL model based on educated guessing, based on extracting relevant parts of the source code, or from monitoring data. Because the software architect only has a partial implementation of CloudStore, the software architect will base the model on a combination. Available artifacts of CloudStore will be extracted into a model, which the software architect will then complete based on educated guessing. The software architect must also estimate resource demands; i.e., how components inside of CloudStore's model use cloud computing resources.

## 2.6 Use Analyzer

The software architect uses the CloudStore's ScaleDL model created in Sect. 2.5 for performing scalability, elasticity, and cost-efficiency analyses using CloudScale's Analyzer tool. In the following, each analysis is reported in a separate subsection.

### 2.6.1 Scalability Analysis

In a scalability analysis, the software architect experiments with several cloud resource configurations for the Web Server and for the Database Server.

The software architect's goal is to find a configuration that delivers a 90 percentile response time of less than 5 s with 1000 simultaneous customers for the Pay operation. At the same time, CloudStore's database has to hold 1,000,000 customer items and 1,000,000 book items (cf. Sect. 2.4.1).

Unfortunately, the software architect cannot find a suitable cloud resource configuration—CloudScale's Analyzer predicts SLO violations for inspected variants. Even if the software architect assumes that a more suitable cloud resource will appear on the market in the next 3 years, Analyzer shows that SLOs cannot be fulfilled.

Therefore, the software architect uses CloudScale's Spotter tool to find the root cause of the problem. Spotter points to the locking of the transactional database as the root cause. The software architect therefore decides to use a NoSQL database implementation instead.

After remodeling CloudStore's ScaleDL model with the new database type, the software architect runs the Analyzer again. The software architect finds that CloudStore is able to fulfill its scalability requirements now. However, the total cost for the envisioned cloud resource configuration is quite high, i.e., \$ 100 per hour. Even if the software architect expects prices to drop during the next 3 years, this amount is more than what the software architect expected.

### 2.6.2 Elasticity Analysis

During the scalability analysis in the preceding step, the software architect used a steady-state version of the load. Now, the software architect conducts an elasticity analysis for transient phases. The software architect looks closer at the best-seller book scenario in Sect. 2.4.2 to see if CloudStore is able to fulfill its SLOs.

Again, the software architect focuses on the toughest workload described in the scalability analysis. As described in Sect. 2.6.1, the software architect has decided to use a NoSQL database. Because this database offers autoscaling, the CloudStore model also reflects this important aspect of the Database Server. Particularly, the `Web Server` also offers autoscaling.

The software architect defines scaling-out when average CPU utilization exceeds 70% and scaling-in when average CPU utilization is below 50%. Using this autoscaling rules, the software architect finds that CloudStore is able to fulfill its SLOs throughout the day. This fulfillment holds for the elasticity requirement to handle an increase, within 1 min, from 500 to 1000 simultaneous requests to the Buy operation and from 500,000 customer items to 1,000,000 customer items within CloudStore's database.

### 2.6.3 *Cost-Efficiency Analysis*

In Sect. 2.4.3, the software architect has defined the cost-efficiency requirement that operational cost should be half the cost of the expensive cloud resource configuration. The software architect runs a cost-efficiency analysis with the Analyzer to check this requirement.

The software architect finds that this requirement is violated—no matter which adjustments the software architect makes in the auto-scaling configuration. However, the software architect finds that CloudStore is able to achieve an overall operational cost of 60% of the operational cost of the most expensive cloud resource configuration (assuming we used this configuration the whole time). As a result, the software architect adjusts the cost-efficiency requirement accordingly.

In this cost-efficiency analysis, the software architect has found that the requirements were too optimistic. Adjusting requirements before investing in development efforts is wise since adjustments allow to resolve trade-offs with different stakeholders before continuing the development effort. An alternative to an adjustment of infeasible requirements is to stop the development project, which can, by all means, be a viable option because of unexpected high development efforts and unrealistic expectations.

## 2.7 Use Spotters

After the software architect has realized CloudStore, the software architect may use the two parts of the Spotter tool to identify anti-patterns. The Static Spotter examines static code, while the Dynamic Spotter comprises instrumentation and load generation of a running service. In both cases, the code is reengineered if anti-patterns or root causes are spotted. Spotter comes with a catalog of supported anti-patterns, which is detailed in Sect. 2.10.

Within a test environment, the software architect has now set CloudStore in operation but observes that response times are too high. In other words, CloudStore's SLOs are violated.

Using the Spotter, the software architect first finds potential weak spots in the CloudStore code and fixes them. Afterward, the software architect experiments with different loads derived from the key scenarios. The software architect finds out that connection pooling is a bottleneck and a simple solution is to make the connection pool larger. This solves all problems, and CloudStore's customers are again happy. The software architect is also happy, being the one responsible for the operation of CloudStore.

## 2.8 Realize, Deploy, and Operate System

After having analyzed CloudStore’s ScaleDL architectural model and its implementation, the software architect realizes CloudStore as a service. At this point, the software architect is more confident than without analyses that the service will satisfy its requirements. The analysis also revealed information that assists this realization, especially because automatic code generation can be employed.

After the software architect has set up the operating environment, the software architect deploys the CloudStore. As an extra precaution, the software architect may use the Dynamic Spotter to identify potential issues in the final production environment.

After deployment, the software architect puts the system in operation with real customers. Monitoring is active during the system operation and enables control of system behavior. Monitoring includes collecting measurements for scalability, elasticity, and cost-efficiency parameters. Measurements allow the software architect to revise SLOs, which potentially triggers a rerun through the CloudScale method.

In this step, the software architect may also discover new anti-patterns. These new anti-patterns can then be put into the anti-pattern catalog (cf. Sect. 2.10) used by the Spotter for its automatic detections.

## 2.9 Cloud Computing HowTos

HowTos describe reusable best practices for software architects to design systems in recurring situations. Such situations typically appear in specific application domains and with respect to particular quality properties. In this section, we shed light on the respective HowTos directly or indirectly related to the cloud computing domain, especially with a focus on scalability and elasticity as cloud computing’s defining characteristics. By following these HowTos, software architects can effectively and efficiently create ScaleDL models and realize cloud-aware systems.

Table 2.1 lists the HowTos so far collected in CloudScale’s catalog of HowTos.<sup>2</sup> This catalog covers the application domains of business information systems, cloud computing, and big data (first column). For each of these domains, the table gives the name of the HowTo (second column), its type (third column), and the fostered quality properties (fourth column).

HowTos in cloud computing focus on elasticity that particularly improves cost-efficiency and depends on scalability. Because of this dependency, scalability HowTos from related domains, i.e., business information systems and big data, are required as well.

---

<sup>2</sup>An up-to-date description of the catalog is available at CloudScale’s Wiki page [5].

**Table 2.1** CloudScale's catalog of HowTos

Application domain	HowTo name	HowTo type	Quality properties
Business information systems	3-Layer	Architectural style	Maintainability
	Loadbalancing (Container)	Architectural pattern	Scalability
	Loadbalancing (Component instance)	Architectural pattern	Scalability
	Static content	Architectural pattern	Scalability
	Sharding	Architectural pattern	Scalability
Cloud computing	SPOSAD	Architectural style	Elasticity
	Horizontal scaling (Container)	Architectural pattern	Elasticity
	Horizontal scaling (Component instance)	Architectural pattern	Elasticity
	Vertical scaling	Architectural pattern	Elasticity
Big data	MapReduce	Architectural style	Scalability
	Hadoop MapReduce	Reference architecture	Scalability

In the domain of **business information systems**, the identified HowTos lay the foundations for cloud computing HowTos. The 3-layer HowTo represents a common architectural style to structure a system into three different logical layers: a presentation layer, an application layer, and a data access layer. Each of these layers can only access the respective lower-level layer. Because of this restriction, the system becomes loosely coupled and therefore more maintainable and easier to scale. Other HowTos can be particularly applied on each layer in separation.

The load-balancing HowTo represents an architectural pattern that requires the existence of a proxy (i.e., a load balancer) that distributes workload for improving scalability. Depending on the variant of this HowTo, the load balancer distributes workload either over a replica of a stateless container, e.g., a virtual machine, or a stateless component instance. For example, the load-balancing pattern can be applied on component instances of the application layer if these are implemented without state.

The static content HowTo represents an architectural pattern that separates static content, e.g., images and static HTML files, from dynamically created content. This separation improves scalability because static content requires no state, and thus, it can be easily load balanced and cached.

The sharding HowTo represents an architectural pattern similar to the load-balancing HowTo: it also requires a load balancer to improve scalability. However, instead of requiring load-balanced elements to be stateless, the sharding HowTo separates workload based on the requested data. The data is divided into partitions (so-called shards) such that each load-balanced element is responsible for only

a particular set of shards. Requests to the same data are then processed by the responsible element.

In **cloud computing**, the SPOSAD [6] HowTo is an architectural style to promote elastic and multi-tenant software applications. SPOSAD describes a variation of the 3-layer HowTo that additionally requires stateless component instances on the middle layer [6]. Because they are stateless, these component instances can be safely replicated (scaled out) and load balanced. In SPOSAD, the load-balancing HowTo can therefore be applied on component instances of the middle layer. For multi-tenancy, SPOSAD introduces a metadata manager on the application layer that provides different tenants with tenant-specific information. This information includes tenant-specific user interface elements, business logic, and data from multi-tenant databases.

The horizontal scaling HowTo [7, 8] depends on SPOSAD's architectural constraints and extends the load-balancing HowTo such that the load balancer dynamically adapts the required number of component instances or container replicas to the current workload. While the load-balancing HowTo improves scalability only, the horizontal scaling HowTo improves elasticity as well.

The vertical scaling HowTo [7, 8] requires that computing resources of a single computing node be dynamically (de-)provisioned. For example, a virtual machine allows to dynamically provision higher CPU processing rates and more main memory. This HowTo therefore improves scalability and elasticity without requiring stateless component instances; however, it can only be applied if sufficient computing resources are available.

In the **big data** domain, the MapReduce HowTo is a common architectural style to foster scalability. MapReduce requires that data be processed independently from each other within mapper and reducer functions. Mapper functions filter and sort such data, and reducer functions summarize collected data. Based on data independence, multiple mapper and reducer functions can run in parallel, thus improving scalability.

Hadoop MapReduce represents a technical HowTo because it suggests using Apache's Hadoop, an open-source MapReduce framework. This HowTo can, however, also be seen as a reference architecture [9] that illustrates how to implement the MapReduce HowTo. The reference architecture essentially describes a processing pipeline for the control and data flow of the MapReduce HowTo.

## 2.10 Cloud Computing HowNotTos

Similar to HowTos that provide with best practices and suggested approaches to common problems, we can also identify the most common *pitfalls* for the design and development of cloud computing systems—so-called HowNotTos. Once deployed in production, these bad practices can result in problems related to security, performance, scalability, and others. In the CloudScale method, software architects

can detect these HowTos on the model level and within the realization of the system. This detection can be both manual and automatic using CloudScale's Spotter.

On the conceptual level, it is worth mentioning that our work differentiates from the current state of the art in performance engineering in an important aspect: we distinguish between performance anti-patterns and scalability anti-patterns. Performance anti-patterns are classical issues which hinder a system to perform as expected under a given constant workload. Scalability anti-patterns are those anti-patterns which prevent the system from scaling when the system has to face higher workloads because of an evolving usage profile (cf. Sect. 4.3). An example for a HowNotTo is *Excessive Dynamic Allocation*, as described in Fig. 2.3.

Focusing here on scalability (and thus indirectly also on performance), we can categorize the most usual symptoms and their most common root causes. In Fig. 2.1, we show a taxonomical representation of the symptoms and causes that are available at the CloudScale Wiki [11] documentation, and which we will outline in the following (Fig. 2.4).

**Application Hiccups** refer to a temporary degradation of the system performance, mostly related to the implementation within internal processes of the system such as a garbage collector, online reindexing of a database, or any such maintenance task that has a system-wide impact due to its complexity and/or the need to work on a big portion of the system's information.

**The Ramp** is the progressive degradation of a system's performance associated with implementation problems, most commonly related to unreleased resources (locks, memory objects, temporary database objects, etc.) or related to unfinished processes that prevent the system to regain access to certain resources.

**The Blob** is a design bad practice in which most of the complexity of the system's logic is enclosed in a single functional unit (e.g., a class), not only decreasing the maintainability of the code, but also resulting in many cases of unnecessary traffic (Excessive Messaging), of massive amounts of data (e.g., sending an entire object instead of only the changes) and requests (due to lack of local reference), and often preventable lock contentions due to poor granularity.

The **Empty Semi-Truck** refers to the usage of messages to send very small pieces of information, which results in Excessive Messaging that would be preventable by clubbing similar or related requests into one single request, greatly reducing messaging overhead, latency, and throughput. The most common cases are related to single-row requests instead of a query to obtain a result set, or the individual requests of different attributes of an object instead of the entire object.

In **Excessive Dynamic Allocation**, chunks of memory or other resources are reserved very often, taking a considerable part of the system's working time. It occurs most commonly in object-oriented-language designs in which objects are created and destroyed very often, overloading the system with the costly task of allocating memory and instantiating new objects from scratch. Having a pool of resources marked as obsolete (objects that are not needed any more, but kept in memory to be reinstantiated when needed) can greatly alleviate this problem.

The **One-Lane Bridge** is a common design in which a passive resource (a connection pool, database, lock, mutex, etc.) is needed by different process at the



**HowNotTo Example:** “Excessive Dynamic Allocation”

**Name:** Excessive Dynamic Allocation

**Abstract:** The Excessive Dynamic Allocation occurs in object-oriented software systems when dynamic allocations are needed. Excessive Dynamic Allocation is where creation and destruction of the objects of the same class are frequent and unnecessary.

**Example:** An example in cloud computing is when the same resource (an object composed of several child objects) offered by cloud vendors in a certain service is regularly needed by most cloud users. Rather than creating and destroying this set of objects dynamically on a per-user basis, a set of objects can be pre-created and shared among the users who need the service.

**Context:** In object-oriented software systems, where many dynamic allocations are used for objects. Typically, those allocations occur in frequent behaviors like loop bodies or event handlers for requests.

**Problem:** When a new object is created in object-oriented software systems, the memory used to contain the object must be allocated from the heap (i.e., a sufficiently large chunk of free heap memory needs to be found and reserved) and any code used to initialize all the objects contained in this memory location must be executed. Memory leaks can be avoided by doing memory clean-up and returning the reclaimed memory to the heap, when objects are no longer needed. Performance can be significantly improved by removing the overheads, which are caused by frequent and unnecessary creation and destruction of objects.

**Detection:** One way of detecting this HowNotTo is based on observing memory allocations and identifying reoccurring patterns with high amounts of allocations of objects of the same class.

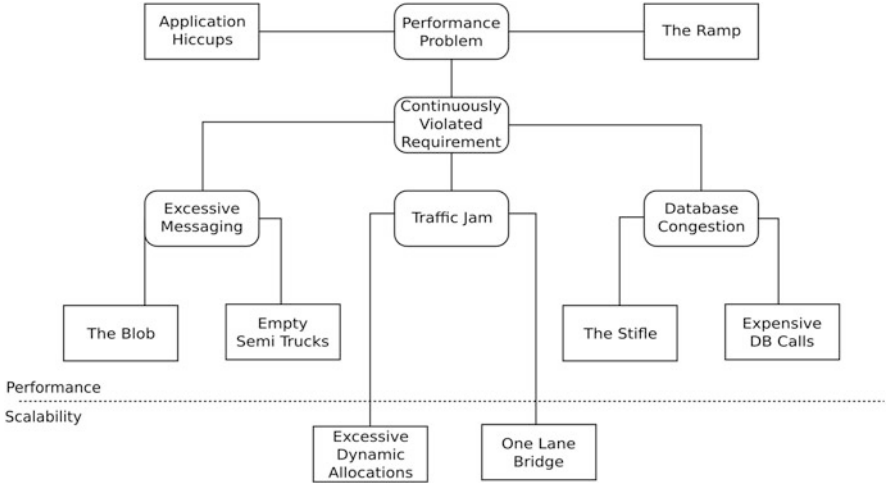
**Solution:** There are two possible ways to solve the problem of Excessive Dynamic Allocation:

- The first is to recycle old objects rather than create new ones every time they are needed. This means allocating and storing a large amount of objects (a pool of objects) in a collection. When new objects are needed, they can be fetched from the pool, and when old objects are no longer in use, they can be returned to the pool. This can be very useful when new objects with relatively short lifespans are created all the time like the objects in the example above. This means we spent a little more time in the initialization of the system caused by allocating objects in the pool, but also reduce the overheads for creating and destroying the same objects all the time. This can help to reduce the problem of memory leaks and the overheads caused by garbage collection.
- The second way is to use sharing instead of creating new objects. An example of this is the use of the Flyweight design pattern, which allows all clients to share a single object.

**Spotter Implementation:** Statically, occurrences of memory allocations (“new”) are detected by identifying parts (a) in which larger memory blocks are allocated (e.g., Arrays) and (b) which are executed often (e.g., in a loop). Dynamically, Excessive Dynamic Allocation can be detected indirectly by leveraging the Application Hiccups detection where the user has to manually analyze whether the hiccups were caused by garbage collection or memory defragmentation.

**See also:** C. Smith [10]

**Fig. 2.3** Excessive dynamic allocation HowNotTo example



**Fig. 2.4** Main synthoms (inner nodes) and their most common root causes (leaves)

same time, becoming the bottleneck for the most important operations of the system by limiting its concurrent processing. A concurrent contention can be alleviated by adding additional passive resources, but the contention point is only moved higher and not resolved. A completely different design approach is needed in order to obtain a really scalable system, though solutions are often impractical, and thus, a good-enough approach is followed.

**The Stifle** is basically an **Empty Semi-Truck** that updates data within the database. For example, if calculating the current age of each employee, we retrieve the birth date, calculate the age, and insert it in that row one at a time instead of using a single SQL statement to update all the rows at the same time.

**Expensive Database Calls** is another design problem that affects the database, this time related to **The Blob**. Complex data queries such as those needed for data warehousing and analytics, or back-up systems, can bring a database’s performance close to zero. This is particularly true for queries that hold locks for longer times (e.g., if consistency of relationships must be preserved), basically preventing operation on most of the database. Night batch processing and parallel warehouse databases are common approaches to solve this problem.

These are, by no means, all the existing symptoms and root causes of cloud computing. A thorough study of these bad practices and common pitfalls can be found in [12].

## 2.11 The CloudScale Method in the Unified Process

After exemplifying the CloudScale method in the preceding sections, this section describes how the CloudScale method fits into existing development processes. In particular, this section focuses on relating the CloudScale method to the Unified Process, as described next.

### 2.11.1 *Unified Processes*

Any kind of software development follows some more or less explicit process steps that transform an initial idea to a working solution. Best practices have been accumulated during the last decades, and currently, there are development processes covering parts of or the whole lifecycle for software products. Some development processes are thin and solve a partial problem, and some of them are more complete. Complete development processes are often termed unified processes. The unified process (UP) is a generic name for a family of process models that are iterative, incremental, architecture centric, driven by use cases, and focus on addressing risks early on. In general, UP defines four project phases: inception, elaboration, construction, and transition. The most recognized unified processes are the rational unified process (RUP) [13] and OpenUP. RUP was created by Rational Software (now owned by IBM) and is supported by commercial tools. OpenUP is promoted by the Eclipse Foundation and is supported by free tools.

Agile development processes have gained traction in recent years. Agility development processes are even more important in cloud-aware development, where the creation of new services is expected to be fast. The Agile Unified Process, as simplification of RUP, has prepared for agile and lean development.

RUP consists of nine disciplines, from which six are related to technical software development: business modeling, requirements, analysis and design, implementation, test, and deployment. Three RUP disciplines are related to executive support: configuration and change management, project management, and environment. In the further analysis, the focus will be on the inception and elaboration phase and the first three disciplines: business modeling, requirements, and analysis and design.

The inception phase is focused on establishing the business case for the system and delimits the project scope. In this phase, all external entities and system roles (actors) must be recognized, and the nature of their interactions described. The business case includes success criteria, risk assessment, and estimates of required resources. The main output of the inception phase is a vision document that contains a general vision of the core project requirements, key features, and main constraints. Additional key outputs are initial use-case models and prototypes.

The elaboration phase focuses on analyzing the problem domain and establishes a sound architectural foundation. To fulfill such an expectation, a wide-enough and deep-enough view of the system is required. Architectural decisions have to

be made with an understanding of the whole system based on the scope, major functionality, and non-functional requirements, such as performance requirements. The main outputs from the elaboration phase are executable architectural prototypes, supplementary requirements capturing non-functional ones, finished use-case models, software architecture descriptions, and potential business models.

### ***2.11.2 Relating the CloudScale Method***

The essential objective of the CloudScale method is to guide software architects by means of an engineering method to develop scalable, elastic, and cost-efficient applications. The CloudScale method provides a framework to build new systems and to analyze deployed systems in operation. Building a new system is focused on system analysis based on a system model and can be done from scratch or by using some existing components. Looking at the described phases of UP, we see that the CloudScale method fits into some elements of the UP, where the CloudScale method performs a deeper analysis of key elements of the system for cloud deployment.

The CloudScale method starts with requirements' collection, with an additional focus on the scalability, elasticity, and cost-efficiency requirements. In the UP, this means that core requirements are extended, with clear concepts for scalability, elasticity, and cost-efficiency requirements. Additionally, system constraints are also clarified. Based on initial requirements and definition of actors, we define the foundation for basic decisions in the CloudScale method: should we develop a system from the scratch, reuse some components, or evolve an existing system into the cloud environment by reusing the existing artifacts of the software system?

When we decide what will be our starting point, we proceed to the elaboration phase in the UP. This phase is correlated with the specification of a ScaleDL model that can be based on the existing software or a completely new model. When applicable, CloudScale's HowTos (cf. Sect. 2.9) provide good assistance for model creation. By constructing the model, we prepare an architecture that can be further analyzed by using Static Spotter and it is the first architectural prototype that can be tested according to scalability, elasticity, and cost-efficiency requirements. An additional possibility is to use some additional external tools and test dynamic behavior of the provided system model. The key supporting elements in the CloudScale method are tools and the provided HowTos (cf. Sect. 2.9) and HowNotTos (cf. Sect. 2.9) that guide system architects in selection of optimal architecture decision and behavior according to requirements. Selecting a good architectural prototype with the CloudScale method creates a very good foundation for the construction phase of the UP.

Analysis of a running system is done in the transition phase in the UP by testing and analyzing system behavior and by collecting critical information about system behavior, with points for potential improvements. The Dynamic Spotter from the CloudScale method supports software architects in this task.

## 2.12 Conclusion

This chapter gives a first overview of the CloudScale method by guiding a software architect through the development of CloudStore. Best practices (HowTos) and common pitfalls (HowNotTos) are also described to assist in the work of the software architect. The overview of the CloudScale method has particularly allowed to relate it with existing development processes such as the Unified Process.

Throughout this chapter, the main benefits of the CloudScale method are exemplified. CloudStore's software architect has been able to analyze scalability, elasticity, and cost-efficiency on the model level, i.e., even before implementing CloudStore. This analysis enabled the architect to resolve any issues early on, which reduced risks of violating SLOs during CloudStore's operation in a production environment. The Spotter tool, HowTos, and HowNotTos additionally helped the software architect in engineering a bullet-proof online shop.

While this chapter only gives a quick high-level overview of the CloudScale method, subsequent chapters provide further details. Particularly, Part III provides detailed step-by-step instructions that explain how software architects can follow the CloudScale method.

## References

1. Q-ImPRESS: Project Deliverable D6.1: Method and Abstract Workflow. [www.q-impress.eu/wordpress/wp-content/uploads/2011/03/D6.1-method\\_and\\_abstract\\_workflow-v2.0.pdf](http://www.q-impress.eu/wordpress/wp-content/uploads/2011/03/D6.1-method_and_abstract_workflow-v2.0.pdf). Visited: 2 November 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., Koziolok, A., Koziolok, H., Kramer, M., Krogmann, K., Küster, M., Langhammer, M., Lehrig, S., Merkle, P., Meyer, 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., Koziolok, A., Koziolok, H., Kramer, M., Krogmann, K. (eds.) *Modeling and Simulating Software Architectures – The Palladio Approach*, 408 pp. MIT Press, Cambridge, MA (2016) [Online]. Available: <http://mitpress.mit.edu/books/modeling-and-simulating-software-architectures>
3. Brataas, G., Stav, E., Lehrig, S., Becker, S., Kopcak, G., Huljenic, D.: CloudScale: scalability management for cloud systems. In: Seetharami S. (ed.) *Proceedings of International Conference on Performance Engineering (ICPE)*, pp. 335–338. ACM, New York (2013). <http://dx.doi.org/10.1145/2479871.2479920>
4. OMG: UML Profile for MARTE. <http://www.omg.org/spec/MARTE/>. Version 1.1, Inspected 11 November 2016, June 2011
5. CloudScale Wiki: HowTos: (2016) [Visited on 12/19/2016]
6. Koziolok, H.: The SPOSAD architectural style for multi-tenant software applications. In: *Proceedings of the 2011 Ninth Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pp. 320–327. IEEE Computer Society, Washington (2011). <http://dx.doi.org/10.1109/WICSA.2011.50>
7. Erl, T., Puttini, R., Mahmood, Z.: *Cloud Computing: Concepts, Technology and Architecture*, 1st edn. Prentice Hall Press, Upper Saddle River (2013)
8. Fehling, C., Leymann, F., Retter, R., Schupeck, W., Arbitter, P.: *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer Vienna, Vienna (2014). [http://dx.doi.org/10.1007/978-3-7091-1568-8\\_1](http://dx.doi.org/10.1007/978-3-7091-1568-8_1)

9. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley Longman Publishing Co., Boston, MA (1998)
10. Smith, C.U., Williams, L.G.: Software performance antipatterns. In: Proceedings of the 2nd International Workshop on Software and Performance (WOSP), pp. 127–136. ACM, New York (2000). <http://dx.doi.org/10.1145/350391.350420>
11. CloudScale Wiki: HowNotTos: [wiki.cloudscale-project.eu/HowNotTos:\\_Anti-Patterns](http://wiki.cloudscale-project.eu/HowNotTos:_Anti-Patterns) (2016) [Visited on 12/19/2016]
12. Wert, A.: Performance problem diagnostics by systematic experimentation. Dissertation, Fakultät für Informatik (INFORMATIK). Karlsruhe Institute of Technology, Karlsruhe (2015)
13. Kruchten, P.: The Rational Unified Process: An Introduction (The Addison-Wesley Object Technology Series). Addison-Wesley, Boston (2003)

Engineering Scalable, Elastic, and Cost-Efficient Cloud  
Computing Applications

The CloudScale Method

Becker, S.; Brataas, G.; Lehrig, S. (Eds.)

2017, XIX, 190 p. 54 illus., 31 illus. in color., Hardcover

ISBN: 978-3-319-54285-0