Jan van Rhijn
6063454
j.vanrhijn@students.uu.nl
Utrecht University

Simulation-based Cost Analysis for Distributed Systems

First Supervisor: J.M.E.M. van der Werf
Second Supervisor: J.M. van den Akker
PwC Supervisor: L. van Dijk

Acknowledgements

This master thesis is the apotheosis of the graduation project of my master Business Informatics at the Utrecht University. This has been created in cooperation with PwC NL Risk Assurance Financial Services at Amsterdam.

This thesis was not created only by me, I want to thank many people for helping and guiding me in this process. Some people I want to thank individually, because they have played a significant role in constructing this thesis. First of all, I want to thank my academic supervisor Jan Martin van der Werf for the inspiration, assistance and feedback. His ideas and constructive feedback really resulted in the thesis as it is. I also want to thank my second supervisor for helping me to graduate. Secondly, I want to thank my daily supervisor Leon van Dijk and my buddies Evert, Michael and Zeliha for the daily assistance at PwC. I also want to thank PwC for accommodating me at their offices for eight months and really making me part of the group.

I also want to thank my fellow interns at PwC for the feedback, guidance and great times at the office. I want to thank the all the colleagues at PwC RAS FS for feeling me part of the group, helping me when they could and to give me inspiration for my thesis. Lastly, I want to thank Martine Verhoef for the final text review.

Abstract

Since 2006 more enterprises are migrating their application stack from an on-premises to a cloud infrastructure. An important difference between both infrastructures is the costing structure. The operational cost of a cloud deployment is more difficult to determine than for on-premises deployments. This is, because of the complex price structure of cloud deployments and the dynamic use of resources. The result of this is that enterprises cannot estimate the operational costs of their application stack when they are moving them to a cloud infrastructure. This is not preferable, because the migration to the cloud is mainly based on financial motivation. The goal of this study is to create a framework which assists enterprises, in estimating the operational cost of their application stack when migrated to the cloud. A tool including a graphical editor and simulation engine is created, guiding software architects to model a software system's architecture and to analyse that, based on operational cost. The user experience of our framework is evaluated by a case study with software architects and the results of the simulation are verified in a series of experiments. The outcomes are promising, and certain interesting opportunities are identified for future research.

Contents

1. Introduction 5

1.1. Problem statement 6

1.2. Research Questions 7

1.3. Relevancy 8

2. Research Approach 9

2.1. Design Science 9

2.2. Research Design 9

3. Running Example 12

4. Dynamic Pricing in Cloud Environments 15

4.1. Introduction to the accounting process within the internet industry 15

4.2. Difference between pricing and costing within this thesis 16

4.3. Pricing theory 16

4.4. Calculators 20

4.5. Results 21

5. Embedding Software Architecture in Cloud-based Environments 23

5.1. Introduction to Software Architecture and Views 23

5.2. Functional View 24

5.3. Deployment View 27

5.4. Mapping between a FAM and Deployment Model 28

5.5. Final Meta-model of a Software Architecture Description 29

6. Simulation Approach 31

6.1. Shift from On-premises to Cloud Infrastructure 31

6.2. Determining the Workload of a Software System 33

6.3. Simulation Approach 37

7. Proposed Solution 38

8. Tool Support 40

8.1. Domain Model 40

8.2. Graphical Editor 42

9. Evaluation of the solution 45

9.1. Evaluation method 45

9.2. Experiments 46

9.3. Case study 50

9.4. Conclusion 51

10. Discussion and Conclusion 52

10.1. Findings 52
10.2. Validity 52
10.3. Limitations 53
10.4. Opportunities 53
10.5. Conclusion 54

11. Bibliography 56
12. Appendices 60

12.1. Appendix A 60
12.2. Appendix B 63
12.3. Appendix C 65

1. *Introduction*

Since Amazon Web Services introduced their Elastic Cloud in 2006 (Barr, 2006), more enterprises are migrating their application stack to a cloud infrastructure. This is called cloud migration and is the process of deploying an enterprise's digital assets, services, IT resources or applications from an on-premises to a cloud infrastructure (Pahl, Xiong, & Walshe, 2013). In almost twelve years more than 70 percent of the biggest enterprises have already migrated at least one of their applications to a cloud infrastructure (IDG Cloud Computing, 2018). The expectation is that in 2022 all enterprises have adopted cloud technology in their application stack and the cloud will dominate enterprise IT decisions (Pettey, 2019). This also reflects in the cloud budgets of enterprises; the average budget of $1.6 million in 2016 has increased to $2.2 million in 2018 (IDG Cloud Computing, 2018). The total cloud services market is expected to grow 17.5 percent in 2019 to a total of $214 billion, up from $182 billion in 2018 (Costello, 2019).

There are several feasible motivations for enterprises to migrate their application stack to a cloud infrastructure. These motivations may be categorized into ease-of-management, availability of new technology and financial aspects (Pahl, Xiong, & Walshe, 2013). A cloud infrastructure takes away a lot of responsibilities for the user, like hardware procurement and maintenance. This increases the ease-of-management for the cloud user (Stuckenberg, Fielt, & Loser, 2011). Economies of scale also have impact on the wide variety of services offered by the cloud providers, which is a considerable advantage for cloud users, because expensive technologies become more easily available to them. Two important financial aspects of cloud migration are capital expense savings and operational expense reductions (Tak, Urgaonkar, & Sivasubramaniam, 2011). Capital expenditure is a capital outflow at the start and is (linearly) depreciated in order to determine the (monthly) costs. This contrasts with operational expenses, which is variable and based on the use of resources each month (Watts & Hertvik, 2018). Because of the pay per use pricing model of cloud services, cloud consumers can reduce their operational expenses.

A cloud migration often takes place from an on-premises to a cloud infrastructure. For this thesis two main differences between those are important, these are: the complexity of the price structure and the variable use of resources (elasticity). The first difference is explained through the ability of the cloud provider to set several pricing capabilities to determine the price of their services, such as server uptime or storage per month capability (Soni & Hasan, 2017). The result of this, is that each cloud resource can have a different set of capabilities, hence the process of determining the price could be different for each resource (Ruiz-Agundez & Garcia Bringas, 2011). This makes the price structure much more complex than for an on-premises infrastructure. The second difference is caused by rapid elasticity, which enables a deployment to time-independently scale up or down to maintain a stable performance. This is one of the five essential characteristics of cloud computing. Rapid elasticity is not possible for an on-premises deployment, where scaling would mean procuring and installing of new hardware, which can take days (Islam, Lee, Fekete, & Liu, 2012).

More than ever, a thorough understanding of the cost structure of a deployment is important. Dutta, Peng and Choudhary (2013) investigated the risks of cloud migrations and discovered that hidden costs could be a risk for a successful completion of such projects. Other research by Alvarez, Hernandez, Fabra and Ezpeleta (2015) suggested that economic costs are a main decisive factor influencing migrating applications to a cloud deployment. Not only for migration projects, but also for cost optimisations purposes a clear understanding of the cost structure is essential (Jallow, Hellander, & Toor, 2017).

Determining the monthly operational costs for an on-premises deployment is a straightforward process. The financial department only needs to know the purchasing costs of the hardware and software and divide it by the expected economic lifespan (Patel & Shah, 2005). The output of this calculation is a constant value: the depreciation rate, also called the monthly operational costs. This process is the same for each resource. Unfortunately, this process cannot be used for determining operational costs for cloud deployments. First, there are no purchasing costs anymore because of the shift from capital expenditures to operational expenditures (Armbrust, et al., 2010). Instead, the operational costs are recurring and based on the use of resources. This is also the case for cloud deployments, where the most frequently used pricing model is pay-per-use (Ibrahimi, 2017). As mentioned before, the use of resources is dependent on the workload due to the rapid elasticity. In a real software system, especially with dynamical arrivals of users/visitors, the workload cannot be known beforehand. Only an estimation of the workload of the deployment is possible, which is a difficult task to do (Patel, et al., 2015). Further complicating the process of cost estimation is the fact that the set of capabilities differs for each cloud resource, so the consumption metrics of those capabilities need to be estimated. This makes a cost estimation for cloud migration projects difficult. An example of this is given in section 3.

An important aspect in cloud migration projects is the decision-making process. A software architecture is a useful approach to represent and communicate such decisions and their rationale (Van der Ven, Jansen, Nijhuis, & Bosch, 2006). The architecture of cloud deployments can change rapidly, because of the on-demand self-service (Mell & Grance, 2011). Good tooling is essential for designing a software architecture and should support designing as well as analysing the architecture in an iterative cycle (Perry & Wolf, 1992). Another advantage of using a tool is that it makes it possible to maintain traceability between multiple views, which helps in analysing the impact of iterations between software architectures (Tang, Jin, & Han, 2006).

1.1. Problem statement

Financial aspects, especially cost reductions, are an important motivator for cloud migration projects. To assess whether a cloud migration project is actually inducing a cost reduction, the operational costs of a cloud deployment must be known beforehand. New features added to the system also have a direct impact on the use of cloud resources, which will impact the operational cost. Hence, insight of the operational cost is essential for systems deployed on a cloud infrastructure.

At this moment, there is no framework which aids in the process of understanding and analysing the operational deployment costs of an elastic software system based on its architecture.

The abovementioned lack of a framework is mainly caused by the complexity of the price structure of cloud resources and the lack of prior knowledge of the workload (and thus scale behaviour) of cloud deployments. This thesis aims to fill this gap in scientific literature and tries to combine both in one framework. The goal of the proposed framework is a thorough understanding of the cost structure of a cloud deployment which aids in the decision-making process for cloud migration projects. The framework will be supported by a self-created tool to enable designing and analysing software architectures in an iterative cycle. First, the price structure of cloud resources is investigated to create an overview of the required metrics and to understand the process of determining the costs of a cloud deployment. Next, the behaviour within cloud deployments is investigated and a method for estimating the workload, based on cloud specific characteristics is proposed. Calheiros, Ranjan, Beloglazov, De Rose and Buyya (2010) suggested a simulation-based approach for assessing the use of resources in cloud resources. This requires a simulation model, which combines the price structures and the behaviour of cloud deployments. In the next phase a tool is created to support the framework and guide the software architects in modelling and defining the software system and automating the simulation. Lastly, the proposed framework is evaluated in a case study to assess the validity of the framework and associated tool.

This study will only focus on estimating the operational deployment costs of cloud deployments. In other words, only the invoices of cloud providers are taken into account. Other factors like maintenance efficiency, reliability or development costs, which also have an impact on operational costs, are omitted.

1.2. Research Questions

Based on the problem statement the following research question is defined:

Research Question: How to improve the process of understanding and analysing the operational costs of an elastic software system based on its architecture.

The first difference between on-premises and cloud deployments is the increased complexity in price structure. The impact of this increased complexity on a cost estimation will be investigated in the first research question. Based on the findings of this investigation, the important capabilities and consumption metrics will be identified. This results in the first research question:

SRQ 1 What are the constituents of pricing in cloud-based environments?

In this research question the findings of the previous question are combined in a software architecture model. The deliverable of this research question will be the foundation for the simulation and proposed framework of this study.

SRQ 2 How to embed software architecture in a cloud-based environment?

In the third research question a simulation approach will be formulated to determine the identified consumption metrics from research question 1. This will be addressed in the following research question:

SRQ 3 How to simulate an elastic deployment to determine usage metrics?

In the last research question the proposed framework and the created tool are evaluated for validity and perceived user experience. Validity is evaluated with a series of experiments and the user experience will be evaluated by a case study with software architects.

SRQ 4 How does the proposed solution perform?

These sub research question will answer the main research question, where we want to improve the process of defining and analysing software architectures.

1.3. Relevancy

This section describes the contribution of this study to the scientific community and society.

1.3.1. Scientific contribution

As indicated in the problem statement, there is currently no approach for estimating the operational cost of a cloud deployment. This thesis tries to fill this gap in literature, but also aims to deliver a self-created tool which can be used for other research. For simulating a cloud deployment and estimating the operational costs, several research areas, such as costing and pricing theories, performance engineering, simulation techniques and software architecture practices, are combined.

1.3.2. Societal contribution

Software architects can use the proposed framework to define their software systems and to automatically estimate the operational costs of their software systems beforehand. This framework also enables enterprises to audit their invoices from cloud providers. At this moment they have to trust their cloud provider, because it is difficult to understand the structure of the operational cost of their software systems. Software architects can also use the proposed framework to compare different architectures in terms of cost efficiency between cloud providers or marginal costs of a new feature implementation.

1.3.3. Thesis overview

In the following section, we discuss the applied research design for this thesis. In section 3 we introduce the running example. The running example will be used throughout the entire thesis for clarification. In section 4 the first research question is discussed; this consists of a scientific literature research and a grey literature research to identify pricing capabilities. In the next section the differences between on-premises and cloud infrastructures are investigated and based on that investigation, a simulation approach is formulated. The outcomes of the last two sections are combined into a simulation model, which is discussed and proposed in section 5.5. Following that section, we propose our framework and in section 8 we create a tool to support this framework. In section 9 we evaluate the simulation model and the created tool. This is followed by a discussion of the findings and a conclusion in section 10.

2. Research Approach

This section describes the research approach used for this study.

2.1. Design Science

This thesis is performed as a graduation project for the master Business Informatics. As the goal of this study is to create a framework (artefact) based on scientific literature, the design science approach is used. The design science approach of Hevner, March, Park, & Ram (2004) is used as framework for this study. Figure 1 depicts the applied framework.

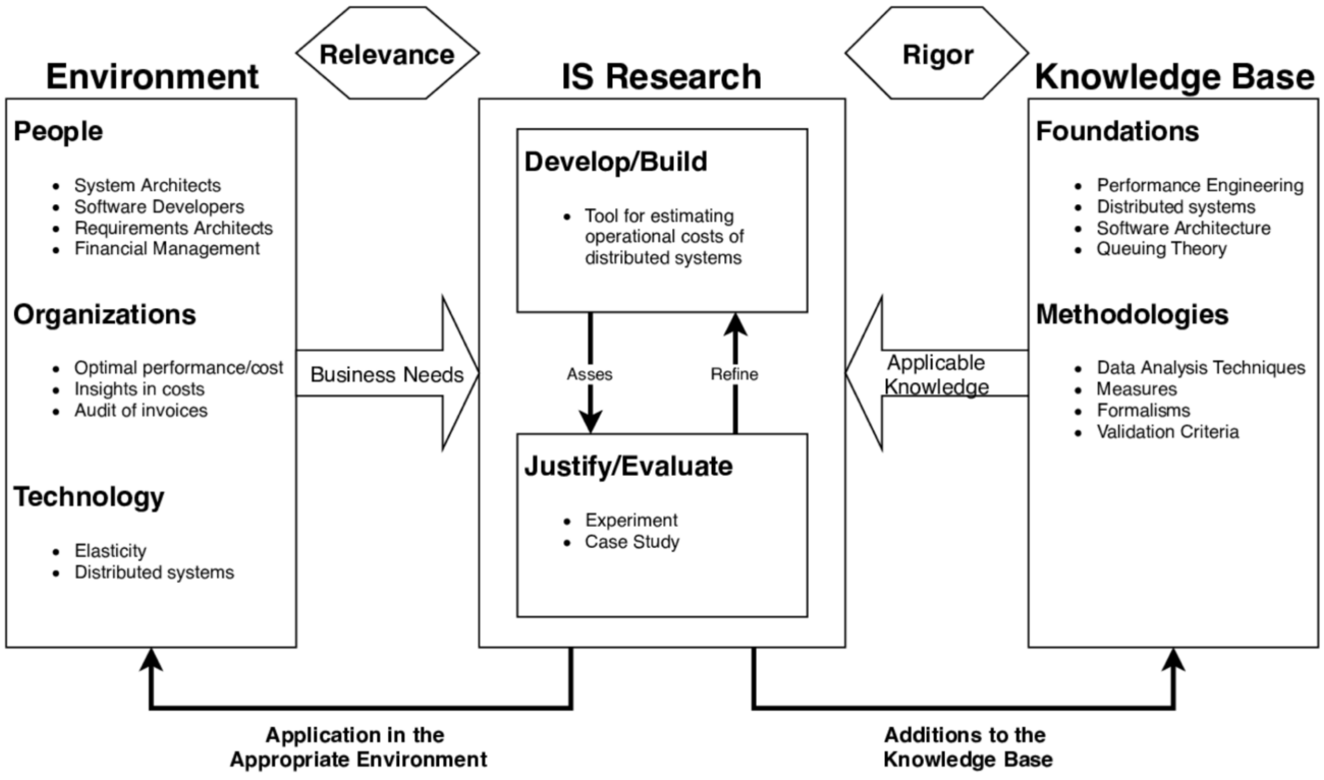


Figure 1 Information System Research Framework Adapted for this Study

2.2. Research Design

Based on the Information System Research Framework by Hevner et al. (2004), five phases of this study are defined. These are:

- Problem statement: describing and understanding the problem. The deliverable is the problem statement.
- Knowledge analysis: identifying scientific knowledge necessary for the solution. The deliverable is an overview of the theory and concepts of the identified knowledge.
- Solution design: the identified scientific knowledge is used to extend the knowledge, so that it can be used for the solution of the problem.

- Tool support: in this phase the solution design is built into a tool, so that stakeholders can use the newly created knowledge. The deliverable is a tool (graphical editor and simulation engine).
- Solution evaluation: based on the tool an evaluation is performed to assess the impact of the solution of the problem.

The abovementioned research design can be applied to the framework of Hevner (2004) and is depicted in Figure 1. The problem statement phase is a result of the Business Needs flow from the model, which is part of the Environment. The knowledge analysis phase is similar to the Knowledge Base, because in this phase Applicable Knowledge is identified for the solution. The solution design phase is the actual Information System Research and can be divided by Develop/Build, which is associated with the tool support phase. The Justify/Evaluate is done in the solution evaluation phase.

2.2.1. Problem Statement

The goal of this phase is to identify the problem statement and associated concepts contributing to this problem. First the problem context is identified and described by researching scientific and grey literature. Once the context is known, further scientific research is performed to understand the problem and identify which concepts are necessary for the solution design. Scientific search engines (Google Scholar, DBLP and ACM) are used to search for scientific literature. Market researches, documentation from cloud providers and cloud-focused blogs are used to search for grey literature. The deliverable of this phase is a problem statement and the required research questions for proposing the solution.

2.2.2. Knowledge Analysis

Based on the outcome of the problem statement phase, six research areas are defined:

- Pricing and Costing Theories for Internet Applications
- Functional Architecture Modelling
- Distributed Systems Modelling
- System Performance Engineering
- Cloud Elasticity/Adaptive Software Systems
- Software (Architecture) Simulation

From each research area, an exhaustive search is performed with the selected search engines: Google Scholar, DBLP and ACM. Based on this search, an unstructured literature study is performed to acquire feasible research papers. Each paper is reviewed by the amount of citations, publication date and publication journal. The goal of this phase is to obtain an exhaustive understanding of the relevant research areas and to use this knowledge to be able to propose a solution in the next phase. The deliverable of this phase is the acquired knowledge and an overview of papers used for this thesis.

2.2.3. Solution Design

In this phase the acquired knowledge is used to design a solution for the problem statement. The solution design is based on the outcomes of the first three research questions. The solution design of this thesis is a step-by-step framework, which will guide a software architect in the process of defining a software architecture description. The framework also enables software architects to analyse the cost structure of a software system based on its defined software architecture. The proposed framework is also the deliverable of this phase.

2.2.4. Tool Support

The solution design formulated in the previous phase will be a tool for software architects where they can use and work with the proposed solution. There are two reasons for the creation of this tool. The first reason is because it allows us to assess the experience of stakeholders of the proposed solution. This is required for validation in the next phase. The second reason is to accelerate the adoption of the solution, because the tool makes it convenient to address the problem statement.

2.2.5. Solution Evaluation

As mentioned earlier, a tool is created to evaluate the solution design. The evaluation will focus on correctness and the user experience of the solution.

1. Correctness of the simulation engine is evaluated by a series of experiments. In these experiments our solution is compared with another, scientifically proven, approach to verify the results of our approach. Another goal of these experiments is to minimize the probability of bugs or errors in our code.

2. User experience is validated through a case study with software architects. In these case studies we evaluated how the software architects use the solution and if our solution fulfils their needs. The outcome of these case studies can be an inspiration for new features.

3. *Running Example*

To clarify and having a better understanding of the problem statement and proposed solution a running example is used for this thesis. In this section the context of the running example is discussed. The E-voter example is used, because it is a well-known example for students who followed the Software Architecture course at Utrecht University. E-voter is a software system for facilitating elections or votes for different institutions.

There are three main usage scenarios for this system, in Table 1 the characteristics of each usage scenario is listed. Each usage scenario has a different order of usage in time, with university elections with the lowest number of possible voters to national elections with the most voters in the shortest timeframe.

Table 1 Usage Scenarios

Type election	**Possible Voters**	**Duration**
University elections	10.000-50.000	1 week
Major city elections	250.000-1.000.000	3 days
National elections	10mln-100mln	1 day

The system has several users, these are listed below. To become a voter, they first must be a visitor (otherwise the front-page is not served to them).

- Electoral council (management)
- Visitor
- Voter

A representation of the functionalities is depicted in Figure 2. There are three main modules: Processing votes (logic layer), Serve Website (view layer) and Votes Persistent Storage (Data Layer) at last there is a Utilities module. The functionality of Convert URL is to convert an URL to an IP address and Route Traffic is to route incoming traffic to specific virtual machines, this is necessary to enable scaling of the resources and to differentiate between requests for different tiers. Both functionalities are part of the Utilities module. The functionality of the Serve Website module is to serve the front-page to the visitors and keep record of the characteristics of the visitors. These are stored in the Website Metrics feature, the front-page is rendered and send back to the visitor by the Render Website feature. The functionality of the Processing Votes module is to process votes and for the Election Council to manage the election. When a Voter votes, the vote will be first authenticated to validate that the ballot is not used multiple times. Then the vote is processed and the voter will be notified, at last the content of the votes are stored in a persistent database. The council is responsible for terminating the election and to process the outcomes of the election, this functionality is represented by the features End Election and Process Results.

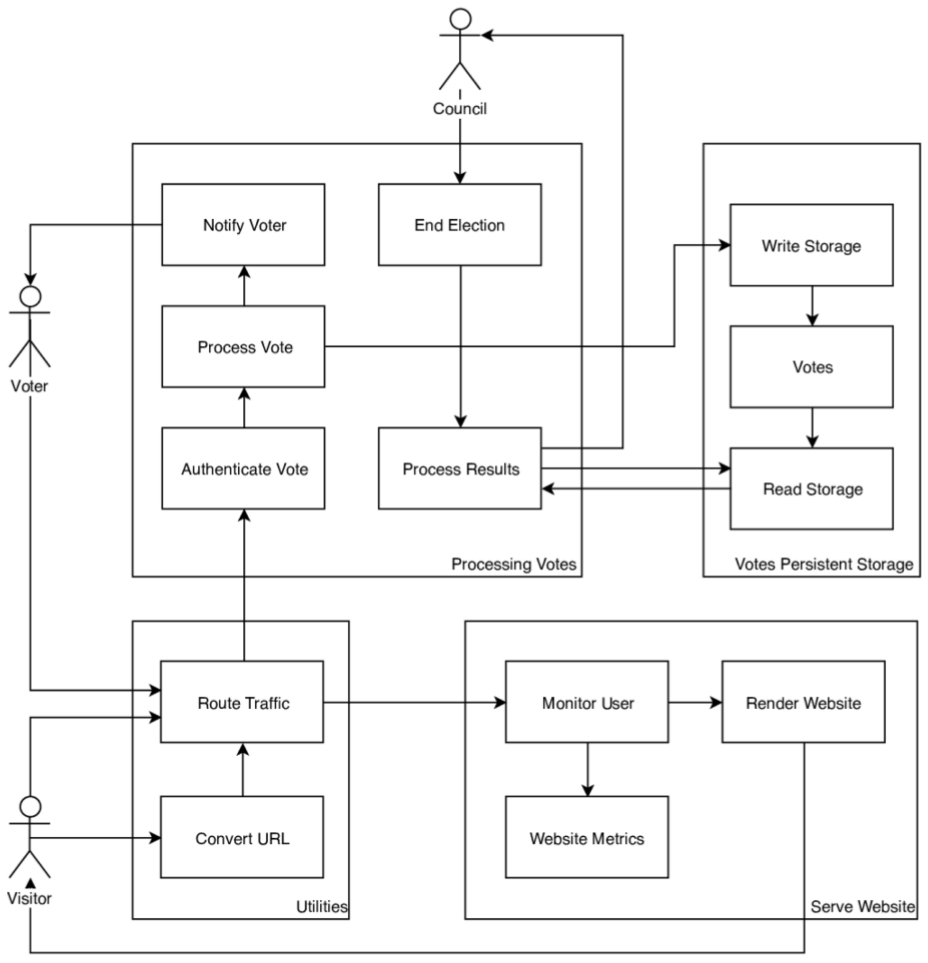


Figure 2 Functional Architecture Model of Running Example

There are four different scenarios: request website, vote, request IP and end election. The sequential order for each scenario is:

- Request website: visitor -> route traffic -> monitor user -> render website and website metrics -> visitor
- Vote: voter -> route traffic -> authenticate vote -> process vote -> write storage -> votes -> notify user -> voter
- Request IP: visitor -> convert URL -> visitor
- End election: council -> end election -> process results -> read storage -> votes - > process results -> council

The system is based on a 3-tier architecture and is deployed on a cloud infrastructure. The view layer, deployed on a webserver, is the website where voters can login and vote. The logic layer, where the votes are authorized and send to persistent storage is deployed on an application server. The data tier, which is responsible for storing and retrieving the votes, is deployed on a database server. The web and application server are deployed as IaaS[1] and the database server is deployed as PaaS[2]. Additionally, there is a Domain Name Server (PaaS), a load balancer (PaaS), an object storage (IaaS) and an external mail service (SaaS). Figure 3 gives an overview of such deployment.

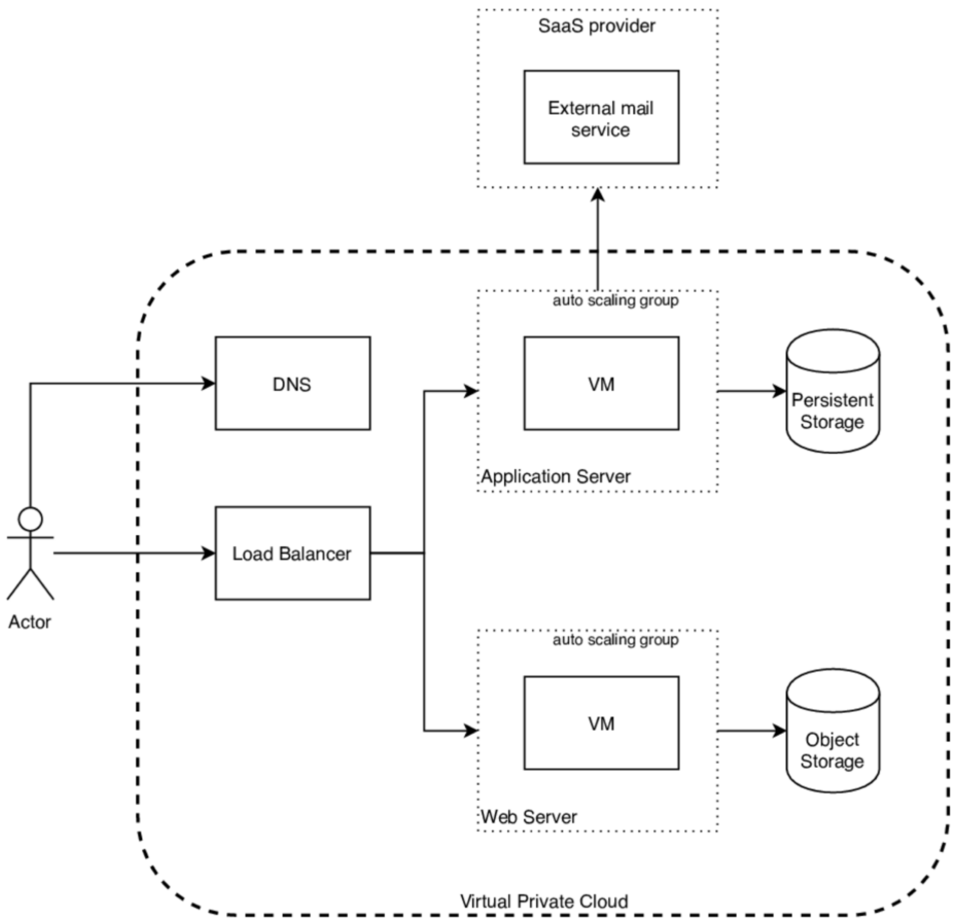


Figure 3 Deployment of Running Example

The price for a virtual machine is \$0.13/h (application server) and \$0.07/h (web server) per running instance. The price of a load balancer consists of the total data processed (\$0.008/Gb) and the time the resource is available (\$0.1/h). The database server is charged \$0.4 per hour and \$0.6 per Gb storage capacity. The price of object storage is \$0.001 per Gb-month and \$0.05 per million requests. The DNS server is charged \$1.5 per million requests and the external email service \$0.15 per thousand emails send. At last all data out of the VPC is charged \$0.1/Gb and data in is free.

The context provided in this section will be enough to make a realistic cost estimation of the E-Voter software system. Although it is difficult to structure the given data, understand how the costs are determined and how to calculate the usage metrics. In the following sections we use the running example to give answers on these questions. At the end of research question three, we will be able to estimate the operational cost of the E-Voter system by hand.

[1] Infrastructure as a Service

[2] Platform as a Service

4. *Dynamic Pricing in Cloud Environments*

What are the constituents of pricing in cloud-based environments?

The goal of the first research question is to investigate the price structure of cloud resources and identify the main capabilities and its associated consumption metrics. The aim of this research question is to acquire a clear understanding of the realization of the operational costs. This is essential to embed software architectures in cloud-based environments and to formulate a simulation approach in the next research questions. First part of this research question is a scientific literature study to investigate the concepts and methods required to determine the costs of cloud related deployments. In the second part these findings are used to identify the main capabilities, used by public cloud providers.

4.1. Introduction to the accounting process within the internet industry

The process of metering the usage of the organisations' assets and based on that knowledge charging customers money is called accounting. Every commercial organization has some accounting processes in order to make profits. For every industry the accounting process will be different, this is more than the case for the internet industry, because of the exhaustive usage of services metering and automatic charging processes. This enables internet companies to use advanced accounting processes for their business models and also explains the complexity of the price structure of cloud providers. In their research Ruiz-Agundez, Penya and Garcia (2010) developed a taxonomy of the internet accounting process, which is depicted in Figure 4. This taxonomy is developed for the whole internet industry, so some aspects in this taxonomy aren't applicable for cloud providers. Although in general this taxonomy is also applicable for a cloud provider accounting process. The process starts with metering the use of the cloud resources. These records flow to the accounting function which collect and aggregate that information. These session records are used as metrics for defining the price of such resource. This is represented in a pricing function, there is not a consensus in literature for the name of this concept. Other names for this are: rating or pricing policy (Aboba, Arkko, & Harrington, 2000). The pricing function will be explained later, on a more detailed view, because this concept has a big impact on the price structure of a cloud resource. The charging process is calculating the cost of a resource based on the use of it. With this process, the technical values (e.g. GB, IOPS) will be translated into monetary units by applying the pricing function to the session records. At last the charging records will be transformed into a final bill or invoice for the customer. This process is called billing. When a customer pays the bill, is called financial clearing (Ruiz-Agundez, Penya, & Garcia, 2010).

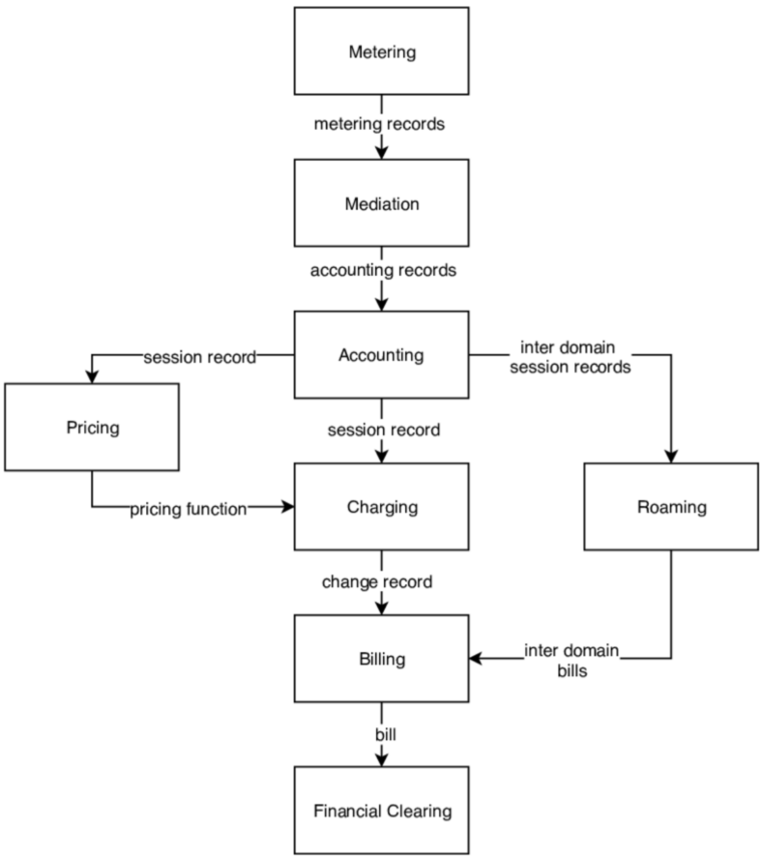


Figure 4 Accounting Process by Ruiz-Agundez, Penya and Bringas (2012)

4.2. Difference between pricing and costing within this thesis

It is important to differentiate between pricing and costing terminology. Pricing, in this thesis, is in the eye of the cloud provider. While costing is in the eye of the cloud consumer of the cloud provider. This is a subtle difference and many researches using both terms interchangeably, which makes it confusing. We start this study with investigating the price structure of cloud resources, because we want to identify which pricing functions and session records are used to formulate the price of a cloud resource. The price is set by the cloud provider; therefore, we speak of the price of a cloud resource. Afterwards we investigate how the use of resources affect the costs of cloud resources. The costs are thus the price with knowledge of the use of resources. Only a cloud consumer can use resources, so we speak of costs in this case.

4.3. Pricing theory

In the accounting process, pricing is represented as the conversion of a session record to a pricing function (Ruiz-Agundez, Penya, & Garcia, 2010). A pricing function consists of a pricing variable (e.g. volume, time) and a pricing coefficient (e.g. price per time unit). The pricing variables are based on the use of the cloud resources (Zseby, Zander, & Carle, 2012). A pricing function can be organized in many ways, each based on different types. In their research Ruiz-Agundez et al. (2012) listed +-20 different types of pricing functions. In general, a cloud provider just uses a few of these function types, those are listed below:

- Time-based function: based on the duration of the services used.
- Volume-based function: based on the volume used.
- Event-based function: based on events in the system.
- Utilization-based function: based on the (maximal or mean) throughput used.

A set of pricing functions is called a pricing scheme. Organizations are free to choose their own suitable pricing functions for their services. Therefore, in theory the set of possible pricing functions is endless (Ruiz-Agundez & Garcia Bringas, 2011).

In the case of a cloud provider a pricing function represent a capability of the specified resource. Like the Google App engine, capabilities are: data storage, number of used instances and outgoing data (Soni & Hasan, 2017). For each pricing function the cloud resource has a capability. Therefore, the price of Google App engine is composed of three pricing functions, which together are the pricing scheme. A pricing function is measured by its own specific consumption measurement metric (pricing variable). In general, this is based on the nature of the resource, like GB per month for storage resources or seconds uptime for computing resources. To convert the metrics to costs, each metric has its own pricing coefficient. When multiplying the measurement metric with the pricing coefficient the cost for that specific capability could be calculated. When the calculated outcomes of all capabilities are summed up the total cost of a cloud resource is determined. A meta-model of such pricing scheme and its relation to pricing functions and pricing model is depicted in Figure 5.

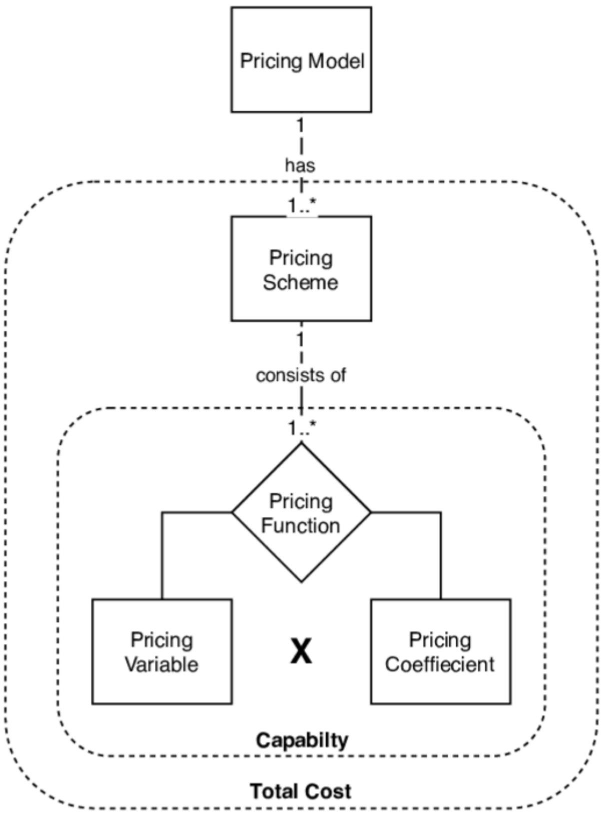
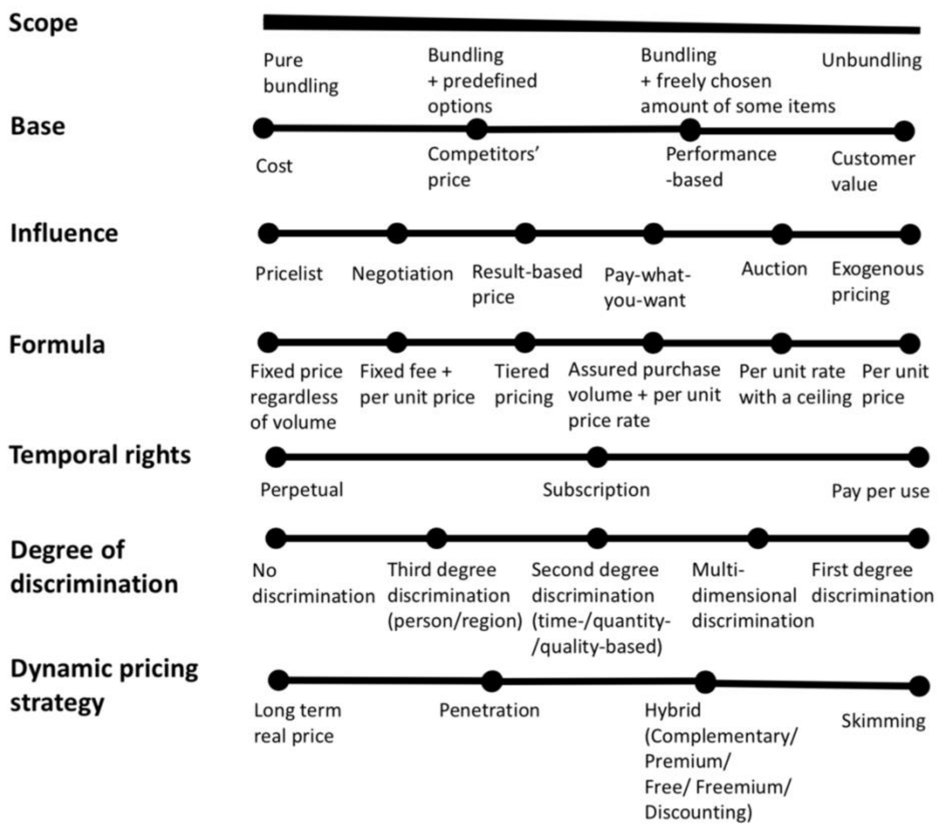


Figure 5 Pricing Strategy Model

At last a pricing scheme of a resource adheres to a pricing model. A pricing model contains the strategy on how the price is specified. To create a systematic approach for determining a pricing strategy, Laatikainen, Ojala, & Mazhelis (2013) updated the SBIFT pricing framework to a cloud context. This framework consists of seven dimensions, each representing pricing strategies offered by cloud providers. This framework is depicted in Figure 6. Some dimensions only influence how the pricing coefficient is specified. These dimensions are: base, influence, degree of discrimination and dynamic pricing strategy. For this thesis we are only interested in the dimensions which specify how the cost of a resource is determined based on the use of it. This is important, because in the next research question we are formulating a simulation approach which aim to determine the use of resources based on its pricing function. Knowledge of those dimensions is essential for determining the costs of such deployments in a simulation. These dimensions are the scope, formula and temporal rights.



Scope
Pure bundling
Bundling + predefined options
Bundling + freely chosen amount of some items
Unbundling
Base
Cost
Competitors' price
Performance -based
Customer value
Influence
Pricelist
Negotiation
Result-based price
Pay-what-you-want
Auction
Exogenous pricing
Formula
Fixed price regardless of volume
Fixed fee + per unit price
Tiered pricing
Assured purchase volume + per unit price rate
Per unit rate with a ceiling
Per unit price
Temporal rights
Perpetual
Subscription
Pay per use
Degree of discrimination
No discrimination
Third degree discrimination (person/region)
Second degree discrimination (time-/quantity-/quality-based)
Multi-dimensional discrimination
First degree discrimination
Dynamic pricing strategy
Long term real price
Penetration
Hybrid (Complementary/ Premium/ Free/ Freemium/ Discounting)
Skimming

Figure 6 SBIFT Framework by Laatikainen et al. (2013)

The scope is an important dimension, because it specifies which capabilities are used for determining the price for a cloud resource. This dimension ranges from pure bundling, where the consumer does not have any influence on the capabilities to use, till unbundling where the customer is totally free to choose which capabilities to use. This will impact the simulation, because to estimate the cost, the used capabilities must be known to simulate such capabilities.

The temporal rights dimension refers to the duration a user can use or access the offered service. This is ranged from an infinite length (e.g. as long as the customer wants) till where a customer pays for each time it uses a service (e.g. pay per use). In general cloud providers offers three kinds of dimensions (Laatikainen, Ojala, & Mazhelis, 2013):

- Perpetual model: the customer is charged for one time only and may use the service as long as they want.
- Subscription model: the customer is charged for a period and have the rights to use the service for that period.
- Pay per use model: the customer is charged for each time they use a certain service. The billing is mostly aggregated for a certain period, otherwise a customer should pay every second, which is not preferable.

The temporal rights dimension influences the simulation approach, because it specifies if a resource must be simulated. With a perpetual model only the presence of the resource is enough to specify the cost. In the case of the subscription model the cost of a resource is known beforehand and therefore simulation on the use of it is not necessary. With the pay per use model, the price is dependent on the use of the resource, a simulation is needed to estimate the cost of that resource.

Another dimension of the SBIFT framework also influencing the cost of a cloud resource is the formula dimension. This dimension is equivalent to the pricing function, which reflect the connection between price and volume. Laatikainen et al. (2013) suggested that there are six kind of formula's structures. Although five structures are used in practice, these are[3]:

- Fixed price regardless of volume: The consumer pays a fixed price regardless of usage. The formula is: P.
- Fixed fee + per unit price: The consumer pays a fixed price regardless of the use and per unit used. The formula structure is: C + X*P.
- Tiered pricing: The structure is the same for the per unit price, although the consumer pays different prices for different levels of use. Mostly a lesser price per unit when use become larger. The formula is: when $X<V$ -> $X*P^1$, when $X>V$ -> $X*P^2$.
- Assured purchase volume + per unit price rate: The consumer pays for a fixed amount of volume used and an overage price is charged for extra consumption of the resource. The formula is: C + X*P.
- Per unit price: The consumer pays for the volume the used. The formula is: X*P.

[3] P is price per unit, C is a constant fee, X is unit used and V is some threshold specified by a cloud provider.

In Table 2, we introduce a mapping of the terms used in pricing theories and in the cloud industry.

Table 2 Relation between Pricing and Cloud Terminology

Pricing terminology	**Cloud terminology**
Pricing scheme	Price of resource
Pricing function	Capability
Pricing coefficient	Price of capability
Pricing variable	Consumption metric

4.4. Calculators

There are many tools for offering insights in the cost structure of cloud deployments. There are roughly two kinds of approaches: 1) a cost calculation or 2) a cost estimation. The first one gives the exact costs of a resource if the utilization is known. This is in contrast with the second approach, where the exact utilization is not known beforehand. Both approaches have their own advantages and disadvantages. A cost calculation is simple and gives the exact cost of an individual cloud resource. Although the results will not be very accurate when a consumer wants to calculate the costs of a complete deployment, because the elastic aspects of a cloud deployment are not considered with a cost calculation (Truonga & Dustdara, 2012). With a cost estimation the utilization of a system will be estimated, and the mean values of these estimations are used as input for a cost calculation. A step further is to simulate the (dynamic) use of the system and estimate the costs based on the simulation output, this will give more accurate results instead of an estimation of the use of the resources (Calheiros, Ranjan, Beloglazov, De Rose, & Buyya, 2010). Drawback of this approach is that a simulation is more complex and process-intensive instead of calculating mean values for dynamic use of cloud resources.

All three major public cloud providers (e.g. Google Cloud Platform, Amazon Web Services and Microsoft Azure) offer a price calculator[4] to determine the cost of their resources. These price calculators provide clear insights which capabilities determine the costs of their resources. Although they lacked some major features like modelling dynamic use of resources and transparent comparing between competitors (Zhang, Ranjan, Nepal, Menzel, & Haller, 2012). Especially a cost calculation makes it difficult to do a realistic and reliable cost estimation of a cloud deployment. Many researches proposed their own solution to fulfil these drawbacks. Like research of Andrikopoulos, Song, & Leymann (2013) who built a decision support system for applications migrating to the cloud. This system includes a cost calculator and enables to select different usage patterns, which is also an approach to model the dynamic use of a software system. The cost calculator in their system is based on RightScale's PlanForCloud solution, which is a free-to-use cost calculator for cloud deployments

[4] These calculators are in the eye of the cloud providers; therefore, it's called a price calculator.

(Hosseini, 2013). Unfortunately, it is not available anymore. Another tool for estimating cloud deployment costs is CLOUD-METRIC, which is part of research from Jallow, Hellander and Toor (2017). It aids developers in the designing phase to estimate costs of IaaS deployments. Although the results are promising it has a few drawbacks, it can only estimate costs on IaaS level and it only support Google Cloud Platform and Amazon Web Services as cloud providers. Cloudorado is a tool for comparing prices of computing, load balancing and storage resources from +20 public cloud providers (Cloudorado, 2011). Although it does not support modelling dynamic use of the system and cannot aggregate the costs for different resources. It is therefore not very accurate as a cost estimation for a complete cloud deployment.

4.5. Results

The deliverable of this research question is an overview of the structure of the pricing schemes for selected cloud resources. We have chosen to investigate the resources from Amazon Web Service, Google Cloud Platform and Microsoft Azure, because these providers are most cited in scientific and grey literature, they also have a share of approximately 75% of the total cloud market (Coles, 2018). Another reason for this decision is their transparent pricing documentation of the offered resources, which is necessary to investigate the price structure. Due to time constraints not all (+-200 each) cloud resources, offered by these cloud providers are investigated. Only the essential cloud resources, necessary for running a standard application (like the running example) are in scope of this study. In further research, more resources can be added to this search.

For seven resources offered by AWS, GCP and Azure the pricing documentation is investigated on used capabilities. Each capability is listed and mapped to each other in order to identify similar capabilities between cloud providers. This was the case for most identified capabilities, which implies that the identified capabilities are cloud provider agnostic. The results of this search are listed Appendix A. The capabilities of all three cloud providers are categorized based on similar properties and are listed in Table 3. The identified capabilities are used in in this research to formulate a simulation approach.

Table 3 Identified Consumption Metrics

Capability	**Definition**	**Consumption metric**	**Unit type**
Server Duration	The time a resource is available.	Seconds/Hours	Time-based
Server Unit	The time a resource is available multiplied by the number of servers deployed in that time unit.	Seconds/Hours	Time-based
Data Transfer In	The data transfer in of the resource.	KB/MB/GB	Volume-based
Data Transfer Out	The data transfer out of the resource.	KB/MB/GB	Volume-based

Data Processed	The sum of the data transfers in and out of the resource.	KB/MB/GB	Volume-based
Storage Capacity	The storage capacity at the end of the billing period.	KB/MB/GB	Event-based
Request	The number of requests to the resource.	Unit/Thousand/ Million	Volume-based

5. Embedding Software Architecture in Cloud-based Environments

How to model a software architecture for a cloud-based environment?

The purpose of this research question is to embed software architecture in cloud-based environments. First part of this research question is a literature study on software architecture. After the literature study we propose an extension of the software architecture meta-model, containing all elements, required for simulating the identified capabilities from research question 1. Subsequently, we discuss a mapping between functionalities and a deployment model. At last we discuss the relationship between our proposed meta-model with the existent standardized meta-model of a software architecture. The proposed meta-model encompasses the views required for the simulation, which will be formulated in research question 3. The goal of the meta-model is to be the foundation of software architecture models for further research. We therefore aim to construct the meta-model as extensive as possible.

5.1. Introduction to Software Architecture and Views

Medvidovic and Taylor suggested that each well-engineered software system should have a software architecture. They stated that the '*architecture of a system is the set of principal design decisions made during development and subsequent evolution'* (Medvidovic & Taylor, 2010). A meta-model of a system's software architecture is defined by Rozanski and Woods (2005) and is depicted in Figure 7. Each system has an architecture, which comprises architectural elements and their interelement relationships. The architectural elements are contained in specific views, which addresses concerns for stakeholders. This is documented in an architectural description, which comprises several views. Currently there are several software architecture frameworks, which incorporate multiple views and different approaches. The most well-knowns are: Kruchten 4+1 View Model (Kruchten, 1995), Clements Views and Beyond approach (Clements, et al., 2002) and the Viewpoint Catalogue of Rozanski and Woods (Rozanski & Woods, 2005). The details of these frameworks are not in scope of this study, but there are some views appearing in all frameworks and are important for this research. These views are a functional architecture view and a physical (deployment) view. A viewpoint governs the architecture view and it contains at least one model which represent some part of the architecture. The meta-model of Rozanski and Woods conforms to the ISO/IEC 42010 standard (International Organization for Standardization., 2011). There exists many modelling techniques (ADL) to construct architecture models, in this thesis we investigate existing ADL's to determine if those meet our requirements.

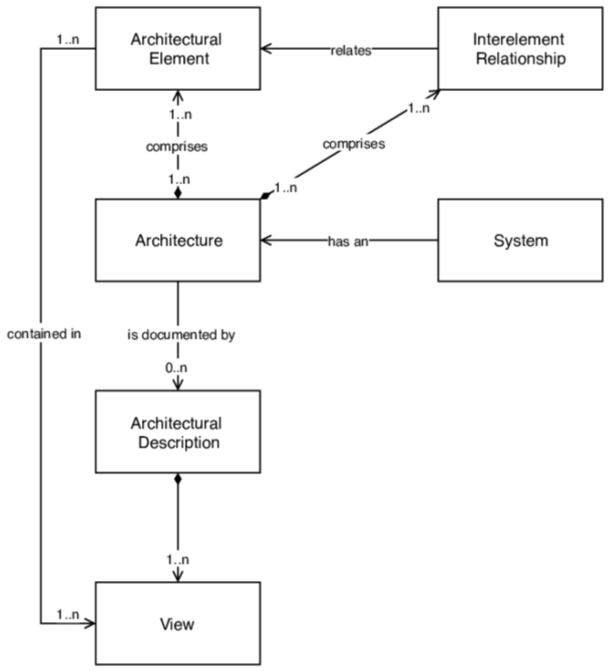


Figure 7 Software Architecture Meta-model

5.2. Functional View

The functional view of a software architecture describes the functional elements and their responsibilities and interfaces. Almost every software architecture description contains a view which describe some functionalities of a software system (Brinkkemper & Pachidi, 2010). A functional view is mostly modelled at the start of a software architecture description, because it addresses concerns for all stakeholders, and it is basically the bridge between requirements and implementation. Concerns are the functional capabilities, external interfaces, internal structure and functional design philosophy of a software system. In practice the functional view is the basis for most other views, therefore the functional view is basically the cornerstone of a software architecture description (Rozanski & Woods, 2005).

Brinkkemper and Pachidi (2010) developed an architecture description language (ADL) for describing the functional view. They called this a functional architecture model (FAM), which is "an architectural model representing at a high level the software product's major functions from a usage perspective, and specifies the interactions of functions, internally between each other and externally with other products" (Brinkkemper & Pachidi, 2010). To make this modelling technique comprehensible for complex software systems, parts of the system can be modelled and visualized independent from each other. We have chosen for this ADL, because it is a straightforward language and it also encompass functional behaviour, which is necessary for our simulation.

5.2.1. Functional Architecture Diagram

A FAM is represented in one or more functional architecture diagrams (FAD). A functional architecture diagram consists of three components: module, feature and informationflow.

- Module: the functionality of the system represented in a hierarchic set of modules. There are two kinds of modules:
    - Composite module: parent of other (sub) modules
    - Atomic module: grouping of features
- Feature: a discrete unit of unique functionality of a system that delivers benefit to the user.
- Informationflow: exchange of information between features.

The functional architecture can be modelled at different levels. The top level of a FAM is the product scope, only modules are used for this level. For mapping functionality to a deployment context, this layer is not enough. The reason for this, is that one module is mostly used for multiple scenarios, which all have different arrival rates. To estimate the workload of a system with multiple scenarios, the level of decomposition must be enough to differentiate between the scenarios and thus the arrival rates. Brinkkemper and Pachidi suggested that a functional architecture is usually modelled in two to three layers (Brinkkemper & Pachidi, 2010). At the lowest level of the FAM each module is a set of features, which is called an atomic module. The meta-model of a FAM is depicted in Figure 8.

A FAM is part of a functional view and consists of modules which can be sub modules of each other. The model always starts with one parent module, which is stored in the containment variable product context. This is the starting point of the FAM and is at the product scope level. A module is an abstract class (dark grey) and the super type of a composite and atomic module class. Only composite modules can contain other modules (composite or atomic), at this level we speak of the module level. An atomic module can only contain features, this is the lowest level of the FAM, which is at the atomic level. A feature can have multiple informationflows, which is stored in the containment flow. An informationflow has a receiver reference to another feature (Brinkkemper & Pachidi, 2010).

5.2.2. Scenario as overlays

In order to introduce behaviour in a functional architecture, the notion of overlays on top of a functional architecture diagram is suggested (Van der Werf & Kaats, 2015). They proposed scenario overlays as a set of interactions between features. A scenario overlay is a use case, which the system can handle. There are two types of behaviour:

- Horizontal behaviour: information flow between modules
- Vertical behaviour: information flow within modules

The software architect is responsible for defining the behaviour in the FAM. This is done by defining a sequence between features for each scenario. In general, this is based on judgement and rational of the software architect. A scenario overlay is based on a use case diagram, which are also modelled in a software architecture description (Van der Werf & Kaats, 2015). For this thesis a software system can handle multiple scenarios and each scenario has its own arrival rate. A meta-model of scenario overlays can be found in Figure 8.

A scenario overlay is another view part of the software architecture description. This view represents the functional behaviour within the system. Behaviour is represented in scenarios, which is a set of interactions. This view can contain multiple scenarios, each representing different use cases within the system. An interaction is implemented by an informationflow, which is part of the functional architecture model. In addition, an interaction can be annotated with properties specific for that scenario, whereas the informationflow can be annotated with general properties. Examples of the former are data transfers and execution properties. Examples of the latter include quality properties like security and privacy. Traceability is maintained, because the interactions within the scenario view is an implementation of the informationflow specified in the functional view. A representation of this meta-model is depicted in Figure 8.

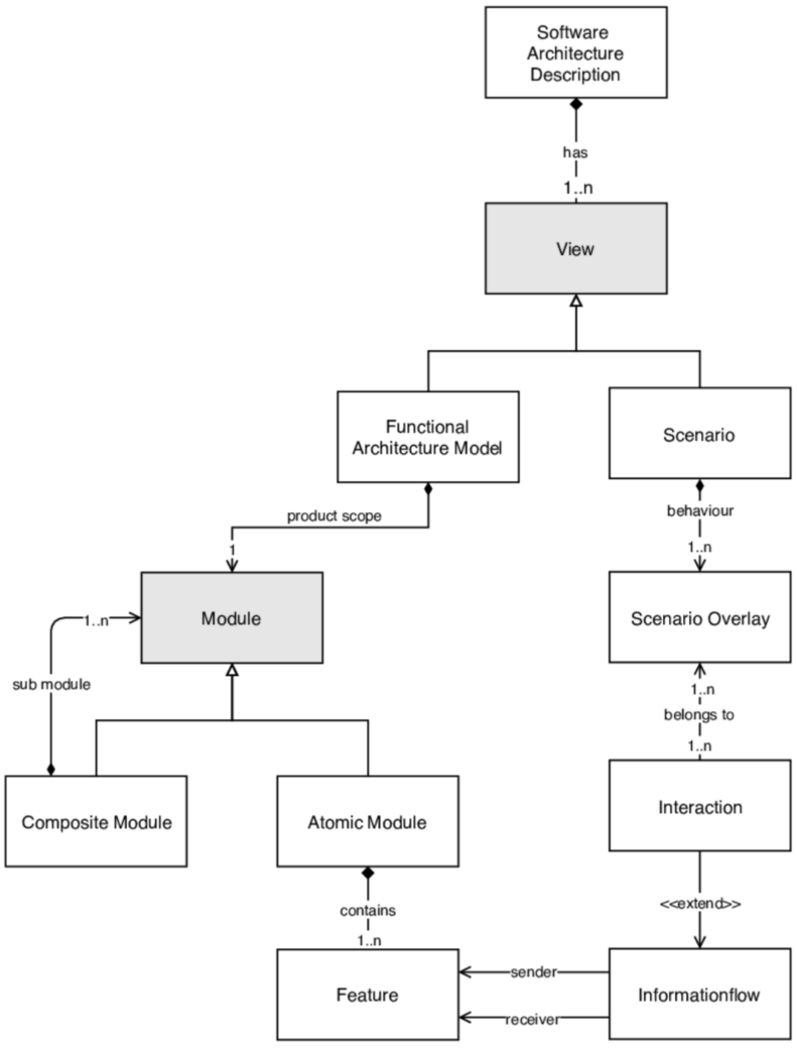


Figure 8 Super Meta-model of a FAM and Scenario Model

5.3. Deployment View

A deployment view is the mapping of software elements to a physical hardware environment. A deployment view is recommended when a deployment environment is not immediately obvious. For example a cloud-based environment, because cloud infrastructures incorporate an additional abstraction layer of hardware (Armbrust, et al., 2010). Concerns of this view are the specification and quantity of hardware, network capacity requirements, and technological compatibility. The most important stakeholders are system administrators, developers and assessors (Rozanski & Woods, 2005).

As mentioned in the problem statement there is no framework combining a cloud infrastructure and costing perspective in a software architecture description. An ADL, which combines both concepts is required for our framework. We therefore propose our own modelling language which incorporate both concepts. Our modelling technique is inspired by AWS Architecture Centre[5] and Azure Reference Architectures[6]. It has the same level of detail as such architecture representations, which is at top level. This makes the modelling technique easy to understand for architects, which will help its adoption. At this level only the cloud resources are modelled, in the future this can be extended to more details like network capacity or performance levels. Currently this is not necessary, because a higher level of detail don't have much impact on the cost analysis of a deployment.

The meta-model of our proposed deployment model is depicted in Figure 9. The meta-model is built around the component class, which represent a resource in a deployment. Each component adheres to a pricing scheme, which a set of pricing functions discussed in section 4.3. The component also has capabilities, which are the identified consumption metrics from research question 1. A capability is represented as the super type to make the meta-model extensible to add more capabilities. To extend the meta-model with more capabilities in further research, the capabilities class is a generalization of the seven identified consumption metrics from section 4.3. Each component contains several services, which are an implementation of the feature class from the FAM metamodel. This advantage of this is that additional properties can be modelled, which are not required in a FAM. Such properties can be service rate or feature type. This also enables traceability between the functional and deployment views and in combination with scenarios, the behaviour in a deployment can be determined. A meta-model of this deployment model is depicted in Figure 9.

[5] https://aws.amazon.com/architecture/

[6] https://docs.microsoft.com/en-us/azure/architecture/

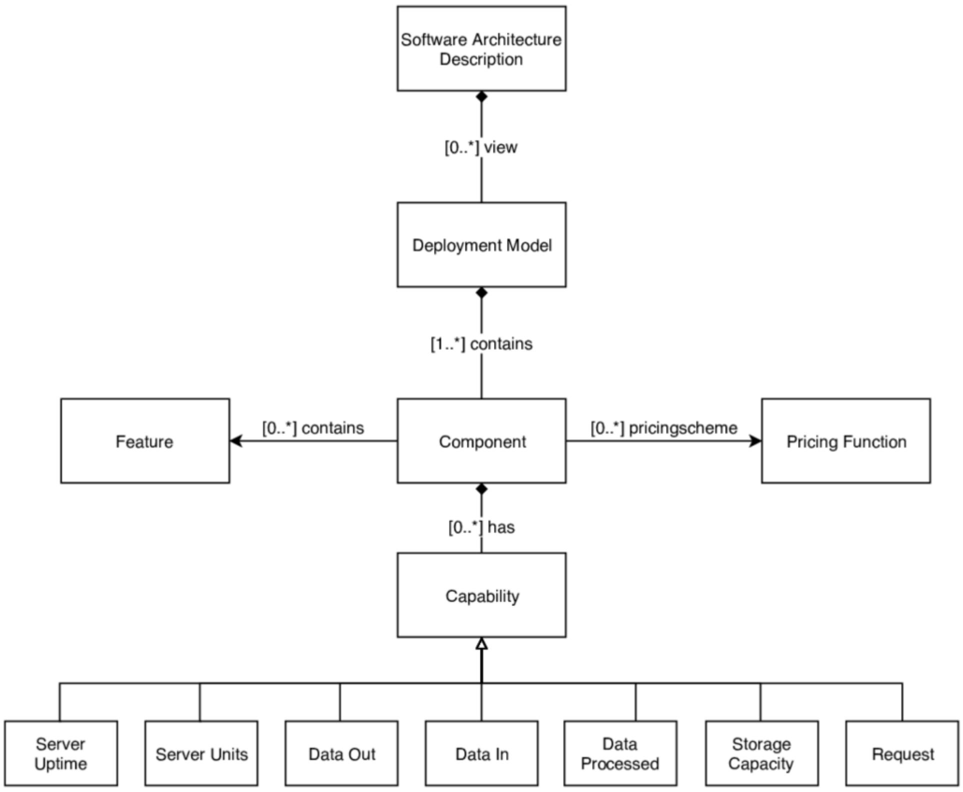


Figure 9 Meta-model of Deployment Model

5.4. Mapping between a FAM and Deployment Model

The aim of this research question is to propose a modelling technique to easily map a functional architecture model to a deployment model. A mapping between both models is important, because scenario overlays and the allocation of features to components represent the behaviour within the deployment of a software system. This can be used to simulate the consumption metrics and making a cost analysis of the software system based on its architecture. In Figure 10 a representation of such mapping is given.

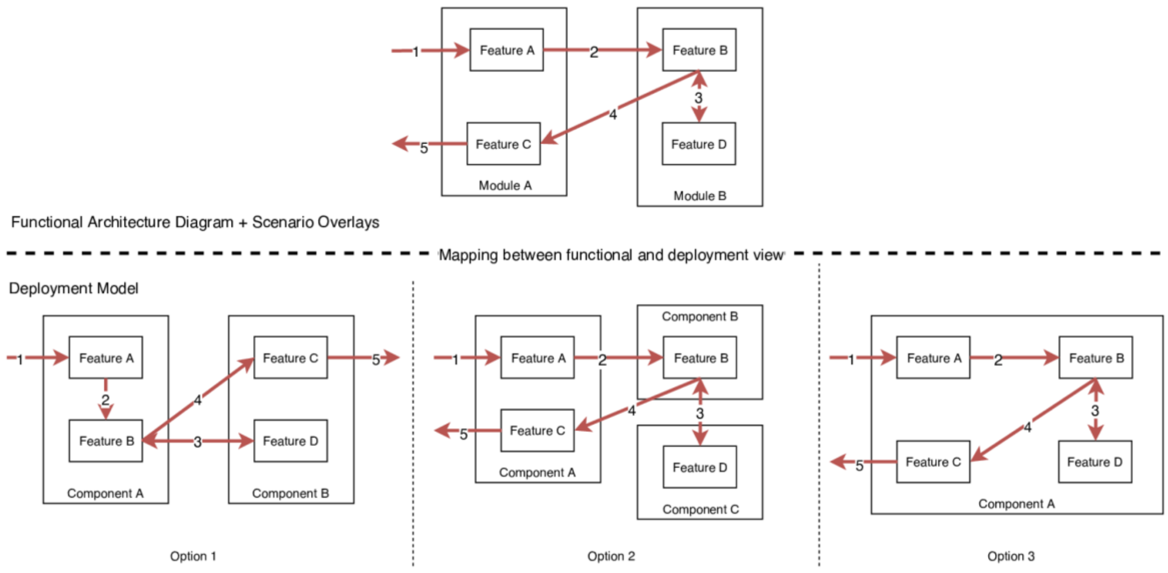


Figure 10 Mapping of a FAM to a Deployment Model

The allocation of functionality depends on the domain knowledge of the software architect. An architect is free to choose such allocation, our model does not require any technical constraints. This is important, because this enables an architect to define multiple setups of a deployment, which can be analysed based on a perspective. For this study we use the costing perspective, but other perspectives, like reliability can also be used. This is future work. In a real deployment there are several kinds of constraints (like technical, physical or legal), which must be considered by an architect. In order to run our simulation, there are two constraints, these are:

- A feature cannot be deployed alone, it must reside in a component. Otherwise capabilities cannot be modelled.
- All features part of the total feature set must be allocated to a component. Otherwise a deployment lacks functionality and behaviour cannot be simulated.

5.5. Final Meta-model of a Software Architecture Description

To compose our simulation model, we merge the meta-models of the FAM, scenario and deployment model to one super meta-model. This is possible because all three meta-models share the same class (feature). The functional, scenario and deployment view are now represented in one meta-model. This enables traceability between views. A high-level overview of the classes and containment variables of this super meta-model can be found in Figure 11.

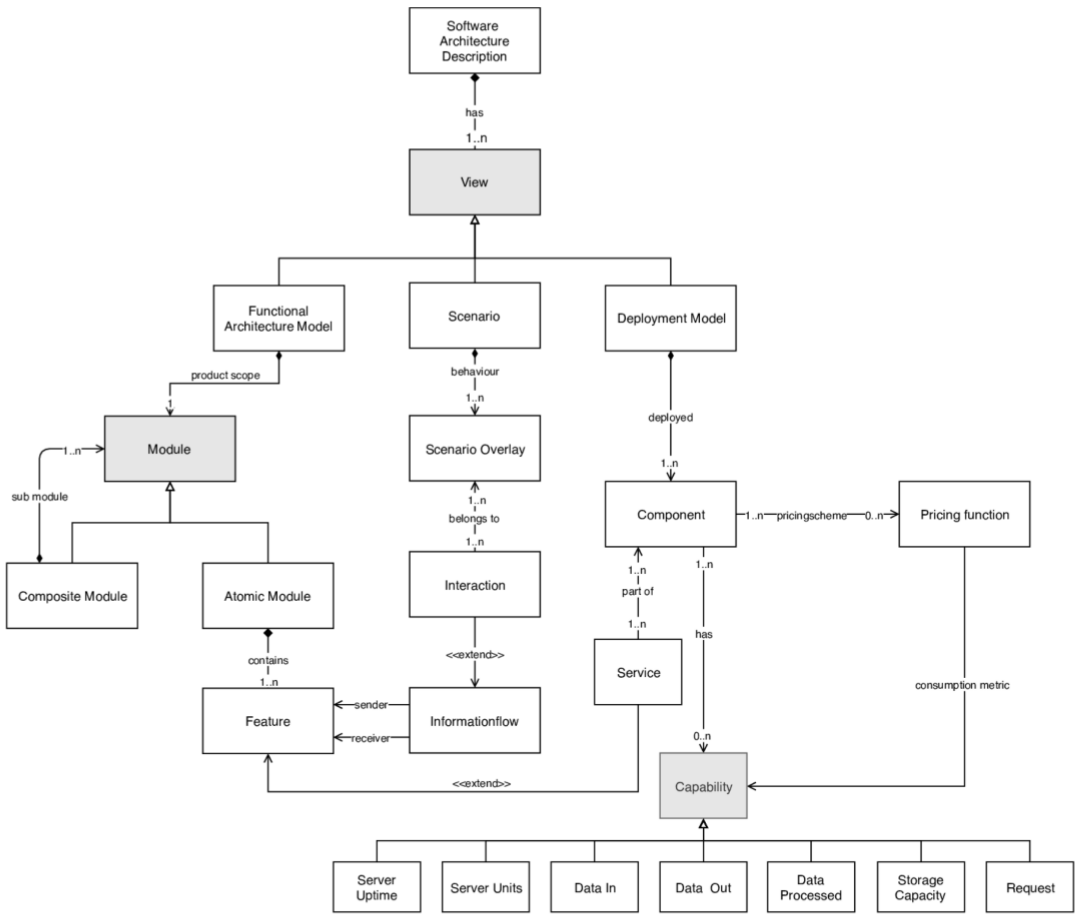


Figure 11 Final Meta-model of a Software Architecture Description

The final meta-model can be linked to the software architecture meta-model from Rozanski and Woods. The nodes in the meta-model are mapped as architectural elements. These architectural elements can be extended with annotations to include quality properties, which are necessary for the simulation. Examples of interelement relationships are the Interaction and Service class, which extends an architectural element and connect two different architectural elements. The classed Pricing Function and Capability are an annotation, because it includes quality propertied required for estimating the operational cost of a deployment. The mapping between both meta-models is depicted in Figure 12. In this figure the relationship between the software architecture meta-model, based on the ISO/IEC 42010 standard, and our proposed software architecture are clearly visible.

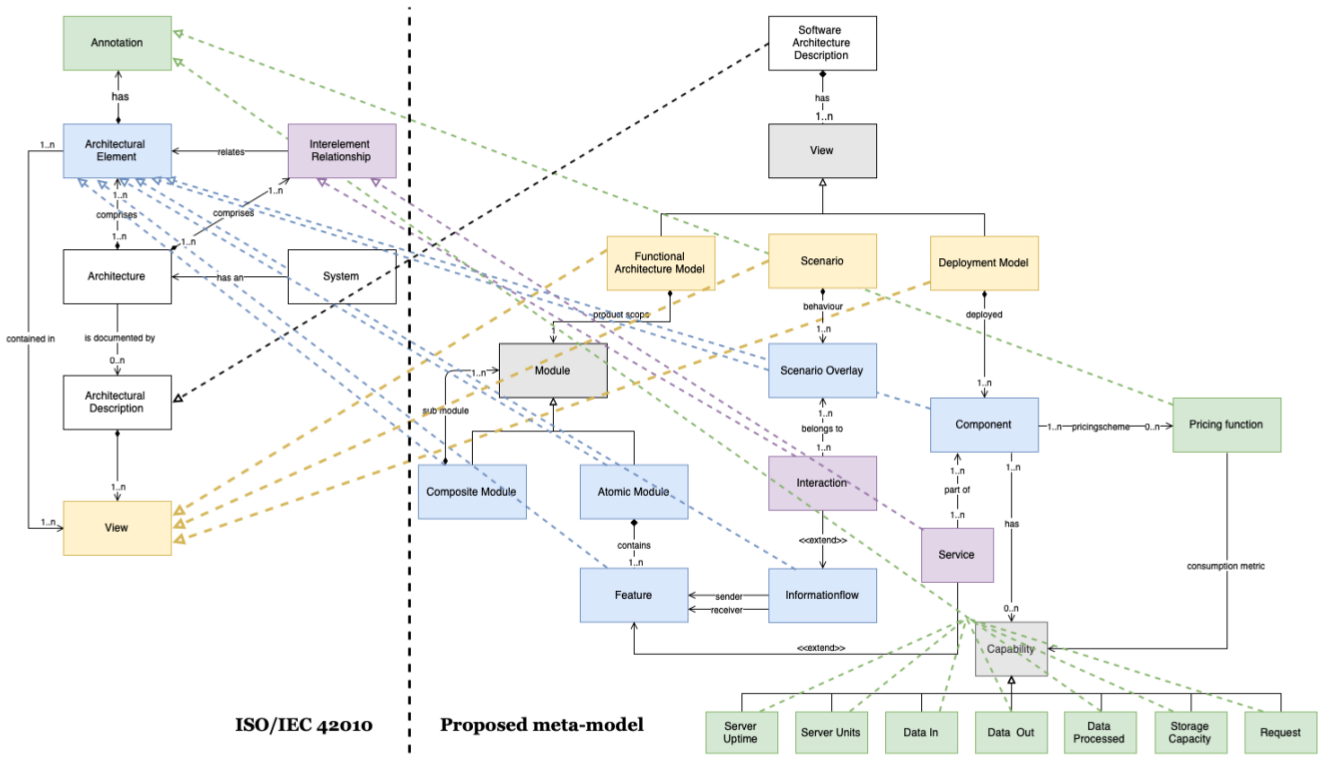


Figure 12 Mapping of Proposed Meta-model to SA Meta-model

6. *Simulation Approach*

How to simulate a cloud-based deployment to determine usage metrics?

The purpose of this research question is to investigate the concepts and mechanisms of an elastic cloud deployment and determine how this can be simulated using queueing theory. The impact of cloud characteristics on the use of resources is investigated and is used to obtain knowledge on how to simulate such deployment. The scientific contribution of this research question are the formulas and algorithms for determining the workload of resources in a cloud context. At the end of this section a simulation approach is formulated.

6.1. Shift from On-premises to Cloud Infrastructure

As mentioned in the problem statement, a paradigm shift to cloud computing is taking place. With an on-premises deployment, the software, technology and hardware are located within the premises of an enterprise or installed on a computer of the user. The procurement and maintenance processes are also the responsibility of the user (Stuckenberg, Fielt, & Loser, 2011). This is different with cloud computing where most of the hardware and software is leased and not the responsibility of the user anymore (Armbrust, et al., 2010). To formalize a simulation approach certain cloud concepts are investigated to understand how the identified capabilities can be determined. These concepts are elasticity and the definition of data transfers.

6.1.1. Elasticity

Scalability is the ability of a system to meet larger workload requirements, by provisioning additional resources, so that a certain performance level is maintained. Elasticity is the advanced form of scaling, with the addition that it can response to workload changes at any time (time-free), this is in contrast of scaling where this process can easily take months (Islam, Lee, Fekete, & Liu, 2012). Another difference is the ability to provision and also releasing instances, while scaling can only do one. This will leverage the pay-per-use pricing model, because an elastic system enables to use resources as efficient as possible.

Elasticity in cloud computing is empowered by auto scaling mechanisms. Such mechanism involves two considerations: 1) how to provision to the required capacity and 2) when to invoke the provisioning process. First consideration is achieved by choosing an appropriate scaling method, these are categorized in:

- Horizontal scaling (replication): the provisioning (scaling in) or releasing (scaling out) process of instances with the same type of instance (Vaquero, Rodero-Merino, & Buyya, 2011).
- Vertical scaling: the replacing or resizing of an IT resource with higher capacity (scaling up) or lower capacity (scaling down) (Vaquero, Rodero-Merino, & Buyya, 2011). There are two approaches to achieve this:
    - Resizing: changing at runtime the assigned resources (memory, storage)
    - Replacement: changing an instance in a more/less powerful one (virtual machine).

Each method is used for a different kind of scaling. Horizontal scaling can only scale virtual machines, while vertical scaling can be used to scale just one capability of a cloud resource (like adding more storage).

The second consideration of a scaling mechanism is when to invoke the provisioning process. Elasticity is about scaling the system according to the required workload at runtime. This is a called dynamic model, where the system can monitor certain metrics and act when thresholds exceeds. Dynamic scaling can be divided in a reactive and proactive model (Coutinho, Sousa, Rego, Gomes, & de Souza, 2015). Static scaling is another scaling method, which is time-based, such method cannot achieve full elasticity, because unexpected loads aren't considered.

- Dynamic model
    - Reactive model: scaling is triggered by exceeding thresholds for certain metrics. The system reacts to workload changes but does not anticipate them (Coutinho, Sousa, Rego, Gomes, & de Souza, 2015).
    - Proactive (predictive) model: the workload of system is forecasted by an algorithm and determine to scale resources before some thresholds are exceeded. Forecasting is usually based on the history of the workload (Coutinho, Sousa, Rego, Gomes, & de Souza, 2015).
- Static model
    - Schedule model: scaling is defined by a schedule. This schedule is based on previous workload patterns. Over- or underutilization of instances is still possible with this model.

The cloud characteristics are well fitted to use dynamic scaling. It also leverages the use of the pay-per-use pricing model, which is often used by public cloud providers. Certain metrics are used to evaluate performance and thus elasticity, these metrics are categorized by Coutinho et al. (2015) in seven categories: allocation, capacity, costs, quality of service, resource utilization, scalability and time. CPU utilization or available memory is often used as performance indicator, because they are easy to monitor and have a direct impact on performance (Coutinho, Sousa, Rego, Gomes, & de Souza, 2015). Elasticity has a major impact on how to determine the use of capabilities. Horizontal scaling and replacing instances influence the number of provisioned servers and thus the server unit capability. Elasticity also influence the data storage capability, because data storage can be resized to store the necessary amount of data at that moment. An outline of elasticity is depicted in Figure 13.

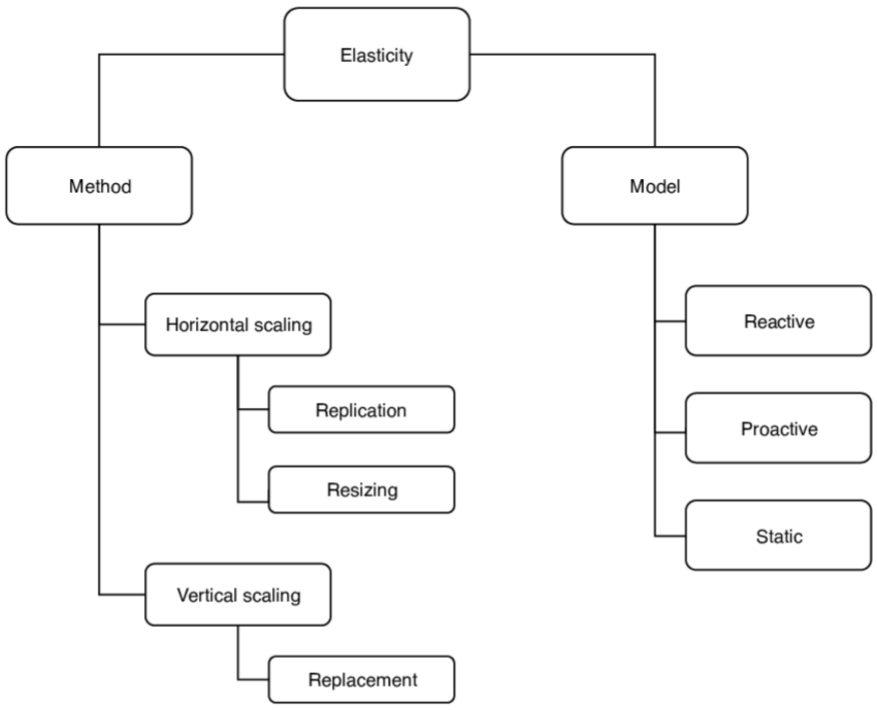


Figure 13 Outline of Elasticity by Coutinho et al. (2015)

6.1.2. Data transfers

Data transfers are an important aspect in cloud computing, especially considering the growth of data intensive applications (Armbrust, et al., 2010). In research question 1 we identified three capabilities involving data transfers; data in, data out and data processed. A cloud provider only counts data transfers between internet and the resource as actual data transfers. Transfers between resources within the same virtual network are not counted as data transfers. Each consumer account with a cloud provider has their own virtual network, where resources can be deployed on (Dillon, Wu, & Chang, 2010).

6.2. Determining the Workload of a Software System

As mentioned in section 6.1.1 the number of provisioned servers depend on the workload at that time. Therefore, estimating the workload is essential to determine the number of servers used for the simulation period. The concept of utilization in queueing theory is similar to CPU utilization (Rose, 1978), which is often used as metric for scaling mechanisms discussed in section 6.1.1. In research question 2 we proposed a modelling technique where we map functionalities to a deployment diagram, the functionalities are represented as a feature. The approach of Klock, Van der Werf, Guelen and Slinger (2017) incorporate the queuing theory as well as multiple features, therefore it is a good fit to use in our workload estimation approach. Within this approach, the utilization rate of a component is calculated based on queuing theory and the service rate per feature.

6.2.1. Queueing Theory

Queueing theory is the mathematical study of queues and is used as a performance modelling technique (Allen, 2014). A computer system can be modelled as a network of nodes. Each function is a node and jobs are moving through nodes in the network to fulfil the specified functionality of the system. This is modelled by a Jackson network, where jobs after completing queue i move to a new queue j with probability $P_{ij,}$ or leave the network with probability $1 - \sum_{j=1}^{m} P_{ij}$. A Jackson network us depicted in Figure 14. A Jackson network require a Poisson arrival process and a first-in, first-out (FIFO) scheduling policy (Walrand & Varaiya, 1980). There are different models which can be used for this theory, for this thesis the M/M/c model is used. With this model the arrivals are determined by a Poisson process, which means it is memoryless (first M). The second M stands for an exponential distribution of the service times. The c stands for the number of servers, which is dynamical because of elasticity (Willig, 1999). When c is known the required number of servers (virtual machines in cloud terminology) can be derived. The service discipline used in this thesis is the is first-in, first-out (FIFO) policy, which is often used (Adan & Resing, 2015).

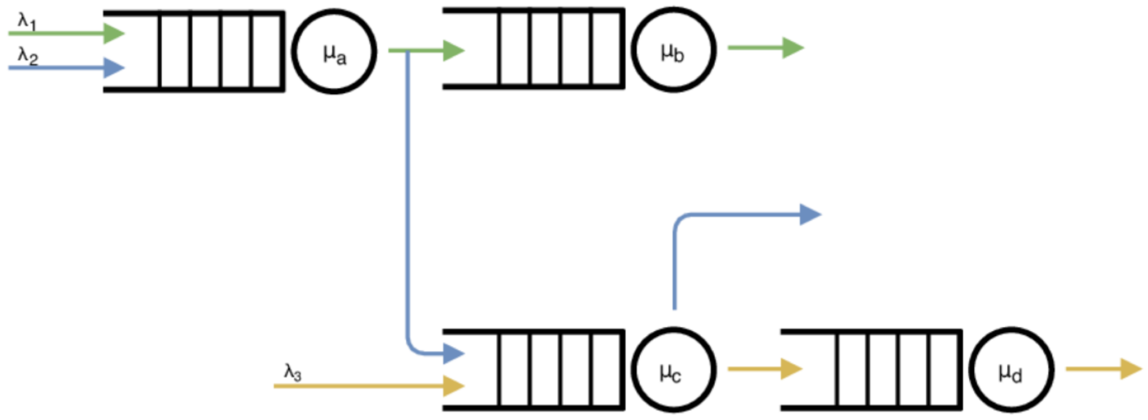


Figure 14 Jackson Network

A queueing node consists of a server and queue element. The jobs arriving at the node are processed by the server. If the servers are busy the job waits in the queue till one of the servers is idle and can process the job. The jobs arriving at the node adhere to a mean arrival rate, which is represented by the symbol λ. A server can handle a mean amount of jobs in a certain time, this is called the service rate and is represented by the symbol μ. A representation of queueing node is in Figure 15.

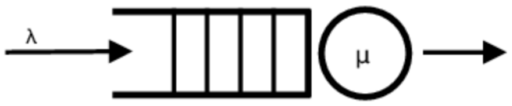


Figure 15 Queueing Node

In the simplest form the utilization of the server is determined by the formula: $\rho = \frac{\lambda}{\mu}$. This is true when there is just one server. To maintain a stable situation, the utilization rate cannot be >= 1.0. A M/M/c model, used in this thesis, can have c number of servers. This means that c will be increased to maintain a utilization rate always < 1.0. The minimal required servers are calculated by the following formula:

Minimal required servers = $\lceil\frac{\lambda}{\mu}\rceil$

Although the minimal required servers are enough to establish a stable situation, it does not indicate anything about the performance. For this the mean queue waiting time is used. The formulas for determining the mean queue waiting time for an M/M/c model are given by Adan and Resing (2015):

Utilization: $\rho = \frac{\lambda}{c\mu}$

Delay probability: $\prod(W) = \frac{(c\rho)^c}{c!}\left((1-\rho)\sum_{n=0}^{c-1}\frac{(c\rho)^n}{n!} + \frac{(c\rho)^c}{c!}\right)^{-1}$

Mean waiting time: $E(W) = \prod(W) * \frac{1}{1-\rho} * \frac{1}{c\mu}$

Based on the queue waiting time, the optimal number of servers (c) can be determined. This is done by increasing the minimal number of servers with one unit, till the mean queue waiting time is below some specified threshold.

6.2.2. Estimation Approach

A component can have multiple features, each with their own service and arrival rate. Just adding up the service rate of each feature makes no sense, because it has different exponential distributions. Klock et al. (2017) coped with this by merging all containing features and calculating a mean service rate for it. Their method can only be applied on horizontal behaviour (between scenarios). In a cloud deployment, vertical behaviour (within a scenario) is also very common. If we use the above-mentioned method, all vertical behaviour should be eliminated first. The sequential order is important; therefore, we propose a three-step algorithm to cope with this. First step is to eliminate all vertical behaviour, second step to calculate a mean service rate and last step to determine the minimal and optimal number of servers.

Step 1: Vertical behaviour exists when multiple features within a component is invoked by the same scenario. When this is the case, we can assume that if one feature is processed in the component, immediately the following feature (part of the same scenario) is also processed. This is forced by the FIFO scheduling policy. This implies that we can add up both service rates (µ) to an aggregated service rate and merge both features. We call this approach the addition method and is depicted in Figure 16.

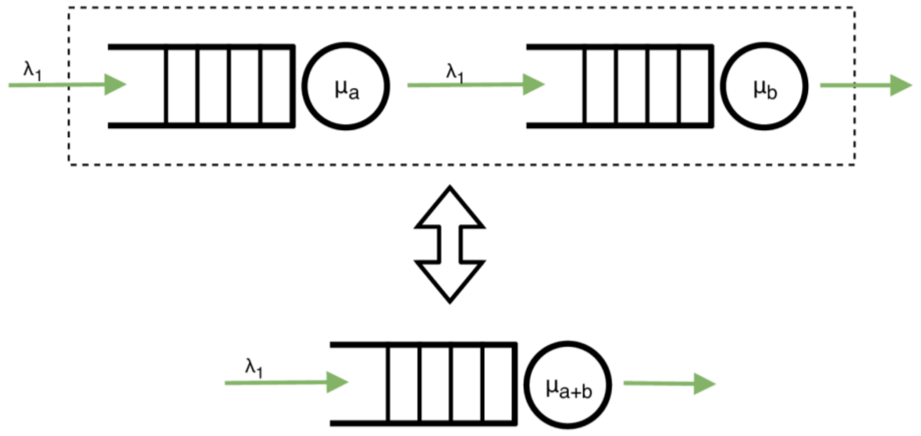


Figure 16 Addition method

The arrival rate does not chance, because the added features are within the same scenario. The new service rate is derived by the following formula:

$$\text{Service rate } \left(\frac{1}{\mu_{a+b}}\right) = \frac{1}{\mu_a} + \frac{1}{\mu_b}$$

Step 2: In this step the mean service rate of a component is determined. This is the same approach used by Klock et al. (2017). In their work, they merge two microservices and determined the service rate of that new micro service by adding the proportional service rate of each microservice, see Figure 17 for a representation of this. This approach can only be applied on horizontal behaviour, because the services are merged together. We called this is the composition method and the formulas are given below:

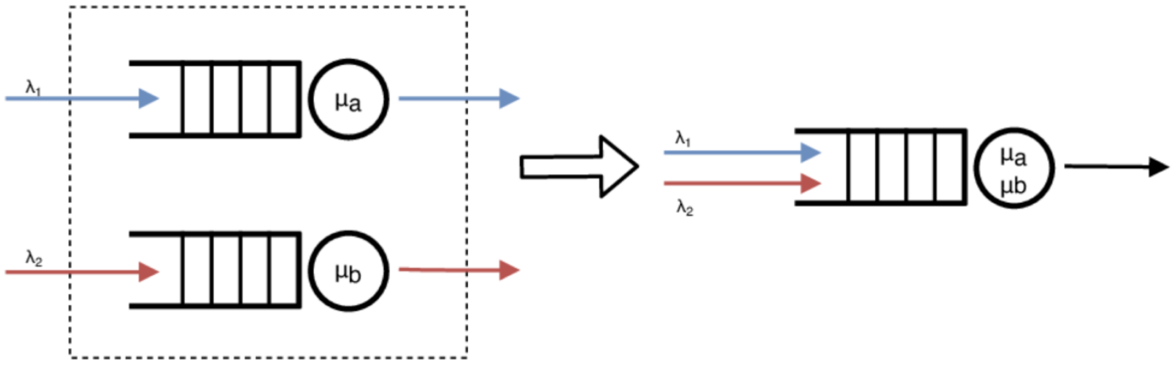


Figure 17 Composition Method

The dashed border represents a component containing two queueing nodes. Left component represents how two scenarios (different colours) invoke both nodes in the component. The formulas of this method are given below and are based on the work of Klock et al. (2017).

$$\text{Arrival rate } (\lambda_{a\&b}) = \lambda_a + \lambda_b$$

$$\text{Service rate } (\mu_{a\&b}) = \frac{\lambda_a}{\lambda_a+\lambda_b}\mu_a + \frac{\lambda_b}{\lambda_a+\lambda_b}\mu_b$$

Step 3: When the new service rate is determined, the minimal and optimal servers can be obtained. The formula and algorithm for this is already explained above.

Figure 18 is a representation of this three-step algorithm is given and an elaboration of this algorithm can be found in Appendix B. The abovementioned formulas are based on the arrival rates of the scenarios. If one of the arrival rates changes the formulas must be recalculated. In a cloud deployment, it is highly likeable that an arrival rate will change after time. Therefore, this algorithm is an iterative cycle, which makes it process intensive when it is applied in a simulation.

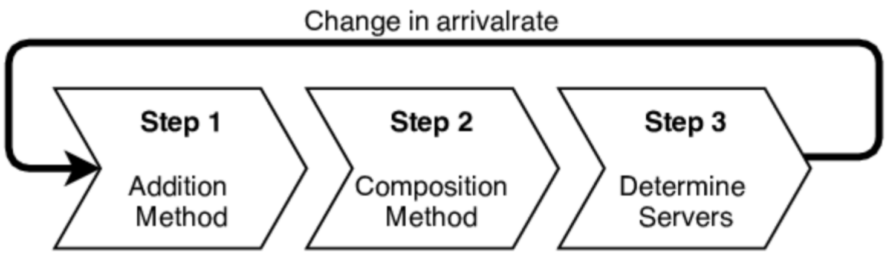


Figure 18 Three-step Workload Estimation Algorithm

6.3. Simulation Approach

In research question 1 we identified seven capabilities required to determine the operational cost of a cloud deployment. In the second research question we investigated cloud specific concepts, which impact how to determine those capabilities and its associated consumption metrics. In this section we formalize a simulation approach, based on the findings on that research. The simulation approach will be used in our simulation engine and is based on the meta-model we proposed in research question 2. The simulation approach is listed in Table 4.

Table 4 Simulation Approach

Capability	**Approach**	**Unit**
Server Duration	Determined by the simulation period	Seconds
Server Unit	Determined by the workload estimation algorithm, discussed in section 6.2.	Seconds
Data Transfer In	Determined by aggregating incoming flows of the component, discussed in section 6.1.2	KB
Data Transfer Out	Determined by aggregating outgoing flows of the component, discussed in section 6.1.2	KB
Data Processed	Determined by aggregating data transfers in and out	KB
Storage Capacity	Determined by aggregating write actions to a storage feature, discussed in section 6.1.1	KB
Request	Determined by aggregating requests to a feature, which are based on the data transfers to the component	Unit

7. *Proposed Solution*

Combining the findings of the abovementioned research questions we propose a framework, which aid a software architect in the process of modelling and structuring functionalities, defining behaviour and architecting a cloud specific deployment for their software system. At last the framework aims to improve the design making process by determining a fitness of the modelled software architecture by making a cost analysis of the deployment. Our proposed framework consists of six activities, divided in two phases:

Activity 1: Define a functional architecture model

In the first activity the requirements are defined as functionalities and structured in modules and connected with informationflows. This activity is similar to the work of Brinkkemper and Pachidi (2010), the modelling technique (ADL), rules and best practices are based on their work. The deliverable is a FAM representing the functional architecture view of a software architecture.

Activity 2: Model functional behaviour with scenario overlays

In this activity the software architecture is extended with scenario overlays to represent behaviour in the system. A scenario overlay consists of scenarios, with an arrival rate property and a sequence of interactions between features. Scenario overlay is discussed in section 5.2.2 and is based on the work of Van der Werf and Kaats (2015). Activity 2 completes the functional architecture modelling part of our framework.

Activity 3: Define a deployment model

In this activity the resources of the software system are defined, and capabilities are allocated to the concerning resources. The deployment context is based on a cloud infrastructure, the deployment model is discussed in section 5.3.

Activity 4: Refer functionality to components

In the fourth activity the functionalities from activity 1 are referred to the resources defined in activity 3. This will maintain traceability between both activities and thus between both views in the software architecture description. When activity 3 and 4 are completed the deployment view of a software architecture description is modelled. After completing this activity, the software system is fully modelled, and thereby the first phase of the framework is completed. In this phase the architecture of software system is defined, based on the super meta-model proposed in research question 3.

Activity 5: Simulation

In activity 5 the software system is simulated, based on the software architecture description model specified in the previous activity. The simulation approach from research question 3 is used to determine the consumption metrics, required for the analysis in activity 6. This activity is quite process intensive, because for each simulation unit the metrics must be recalculated. In the next section we create a simulation engine for interpreting the simulation model and using the simulation approach for automatically calculating the consumption metrics for each capability.

Activity 6: Analyse software architecture

The determined consumption metrics from the previous activity are used to calculate the cost for each capability of each component. Based on the calculated costs of each capability, the total cost of a resource can be determined by aggregating the cost of its associated capabilities. The operational cost for each component can also be aggregated to determine the total operational cost for the whole deployment. This activity gives a detailed overview of the cost structure on three levels (per capability, per component and per deployment). This activity can be extended to analyse a software architecture of multiple perspectives, like performance or reliability. The last two activities are part of the analysing phase.

The proposed framework is a step-by-step approach to guide an architect in designing a software architecture description and analysing the system based on a cost perspective. Currently this framework supports three views, the functional, scenario and deployment view. In further research this can be extended with more views for modelling more advanced software architectures. The proposed super meta-model from research question 2 is the foundation of this framework. Because of this super meta-model, traceability between views is maintained. As mentioned before, the goal of the framework is to analyse a software architecture and based on that findings adjust the model in an iterative manner. This means that activity 3-6 are recursive in our framework, see Figure 19 for a representation of this.

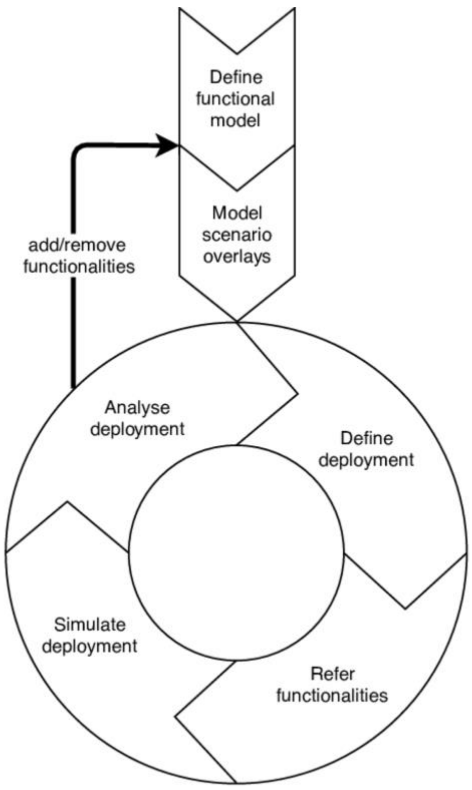


Figure 19 Process overview

8. Tool Support

The proposed framework in section 7 is a first step to formalize the process of defining a software architecture description. One of the main goals of the framework is to maintain traceability between multiple views. This is solved by using one super meta-model for the framework. Another goal is to automate the process of defining and simulating the software architecture description and to make the framework easy and straightforward to use. In order to fulfil these goals, we created a tool to support each activity of the framework.

As mentioned before, the framework consists of two phases; 1) defining a software architecture description and analysing a software system. A graphical editor is created to support the defining phase of the framework. We have chosen to use graphical modelling over text modelling, because a graphical model is easier to interpret and more important, it enables to represent traceability between views on a straightforward manner**.** For the fifth activity a simulating engine is created, which automatically calculate the consumption metrics for each component given a certain simulation period. The simulation engine is based on the formulated simulation approach from research question 3. At last a cost analyser is created to analyse the cost structure of the software system. The super meta-model is built on top of the Eclipse Modelling Framework[7] (EMF), which enable model-based development. The engine and analyser are built in Java as an Eclipse plugin and for the graphical editor the Sirius[8] framework is used.

8.1. Domain Model

The core of EMF is the super meta-model from research question 3. An overview of the EMF meta-model can be found in Figure 20. This model is an extension to the final model of research question 3, because it also includes the properties for each class, which are necessary for the simulation and cost analysis. In section 5.4 a mapping between a FAM and a deployment diagram is proposed. This mapping, in combination with scenario overlays, enable to represent behaviour in a deployment context. For some capabilities this model is not specific enough in order to determine the consumption metrics found in research question 1. We therefore extend our super meta-model with additional properties below:

1. For each interaction a *data* property is added. This is needed to determine the data transfers between the features and on a higher level between components. This extension is essential to calculate volume-based capabilities.

2. For each component a *part_of_VPC* Boolean is added. This property is used to specify if a data transfer is within, inside or outside a private virtual network. This is necessary to determine if an interaction will increment the data in or data out capability, as discussed in section 6.1.2.

3. For each component a *max_waiting_time* property is added. As discussed in research question 2, the optimal number of servers can be calculated based on the max waiting time of a

[7] https://www.eclipse.org/modeling/emf/

[8] https://www.eclipse.org/sirius/

component. To offer more flexibility it is possible to specify for each component the maximum waiting time.

4. For each feature a *feature_type* is added. This is used to specify if functionality is a processing, storage or connector feature, discussed in section 6.3. In the EMF model is this specified by an enumeration (green class), which enforce a feature to have a PROCESS, STORAGE or CONNECTOR type.
    - A compute feature is for functionality which is based on processing power. With this feature-type a service rate must also be specified, because this is used to determine the workload of the component the feature resides in (discussed in SRQ 2).
    - A storage feature is for functionality based on storing data. It treats an incoming informationflow as a write operation. An out coming information flow is treated as a read operation.
    - A connector feature is for anything else. The functionality is basically serving as connection point of a component.
5. For each scenario an arrival rate property is added. The arrival rate is essential, because each interaction is multiplied by the associated arrival rate of the scenario in order to calculate the consumption metrics.
6. For each interaction a *force_receiver_as_connector* Boolean is added. This property specifies if the receiver of this interaction is treated as a connector. A use case for this, is when a feature is sending a read operation to a database and expect data in return. When this property is false, the feature is counted twice for determining the workload of the component.
7. For the capability class the *consumption_metric* property is added in order to specify which consumption metric is used for calculating this capability. Each capability has several unit type properties (*duration_type, data_type and request_type*). The reason for this is that prices offered by cloud providers can differ in unit types. With specifying the unit type an architect does not have to transform each capability to a certain format. This makes appending a capability more straightforward and faster. At last a *billing_type* property is added, which is for specifying if a capability is billed per hour or per second. All unit types are an enumeration (green class), which enforce only variables specified by the meta-model.

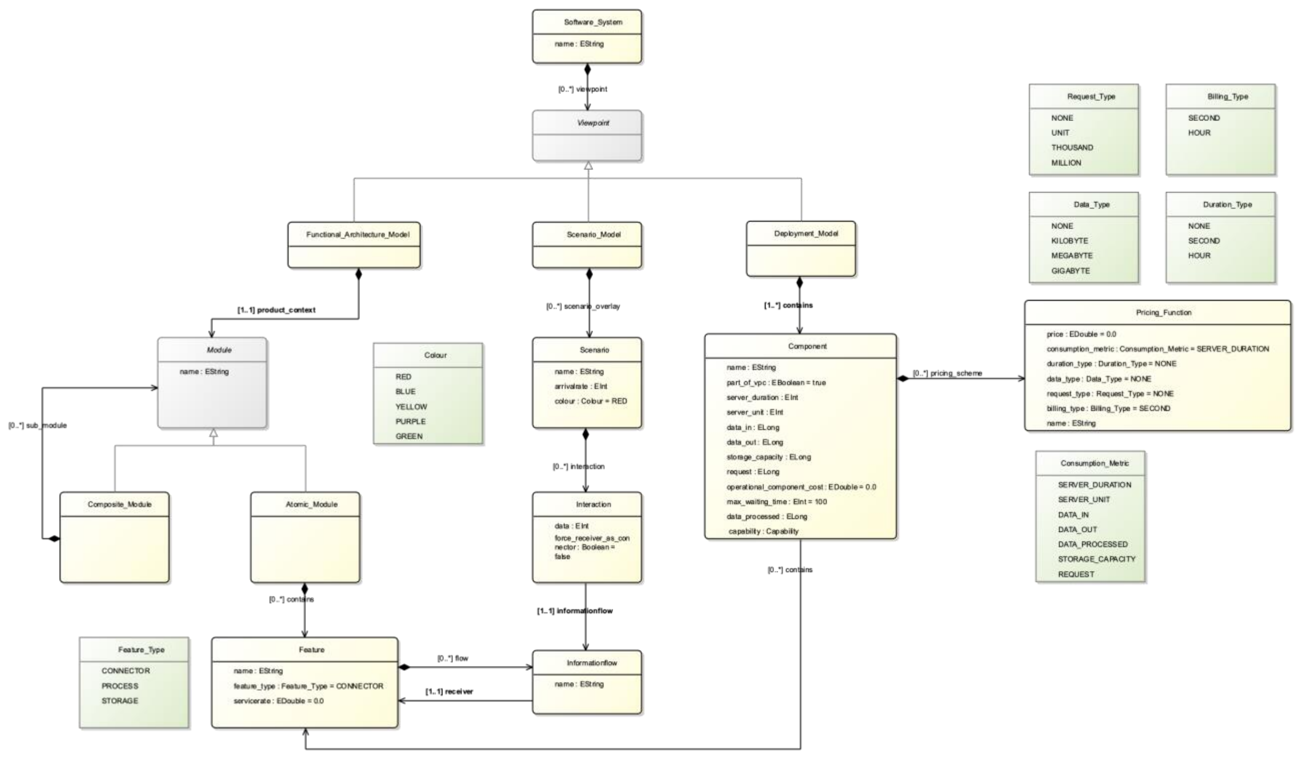


Figure 20 EMF Meta-model of a Software Architecture Description

8.2. Graphical Editor

Based on the EMF meta-model from Figure 20 a graphical modelling tool is defined in the Sirius framework (version 6.2). The editor consists of two layers, each for the functional and deployment view. For the functional and scenario view the representation style from Brinkkemper and Pachidi (2010) is used. As for the deployment view, we defined our own ADL. The tree different feature types are distinguished by a light (connector), middle (storage) or dark (processing) grey node. Components, part of a virtual network, are yellow nodes. Behaviour within the deployment model is represented by a colour for each scenario, which makes it straightforward to distinguish between scenarios. At last a pricing function is represented as a border node on the right of a component. Because the graphical editor is based on the EMF meta-model, consistency between the graphical model and our proposed super meta-model is enforced.

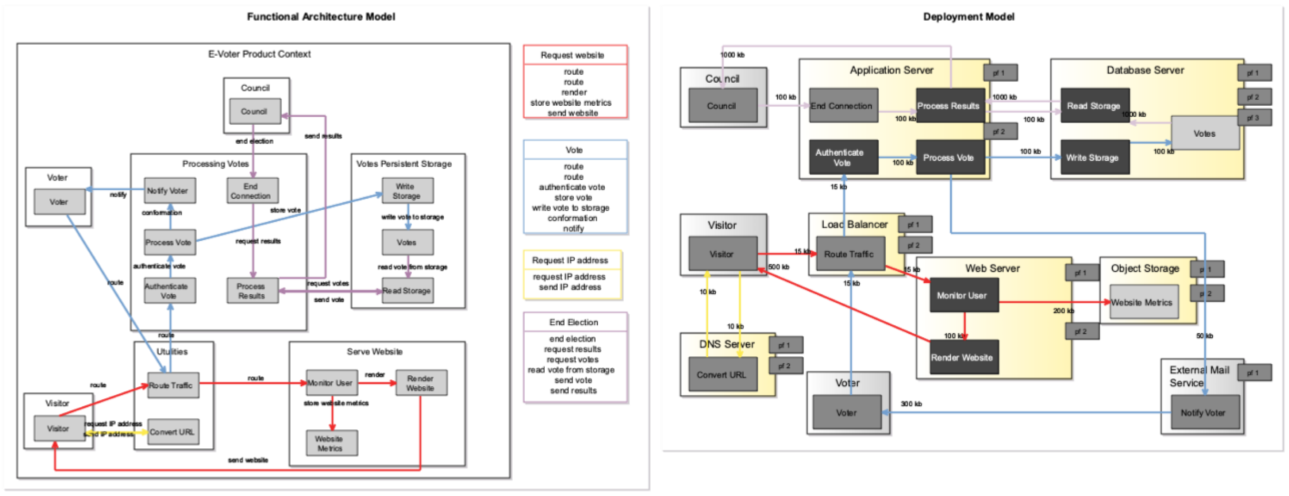


Figure 21 Representation of the Graphical Editor

A main goal is to maintain traceability between views, this is represented in Figure 22. Each feature from the FAM is connected to a feature in the deployment view. The representation of traceability is optional.

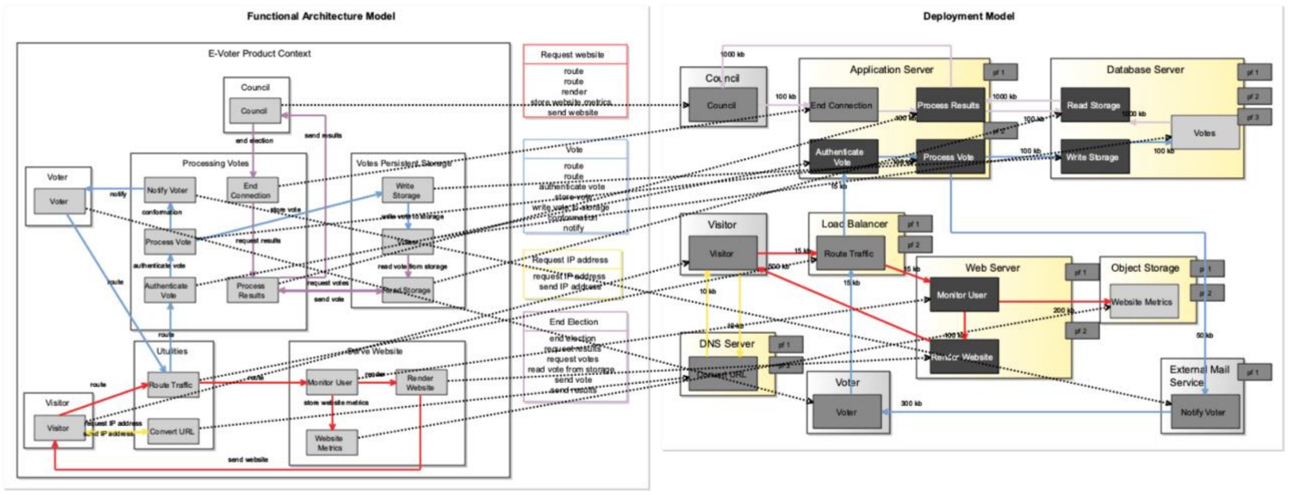
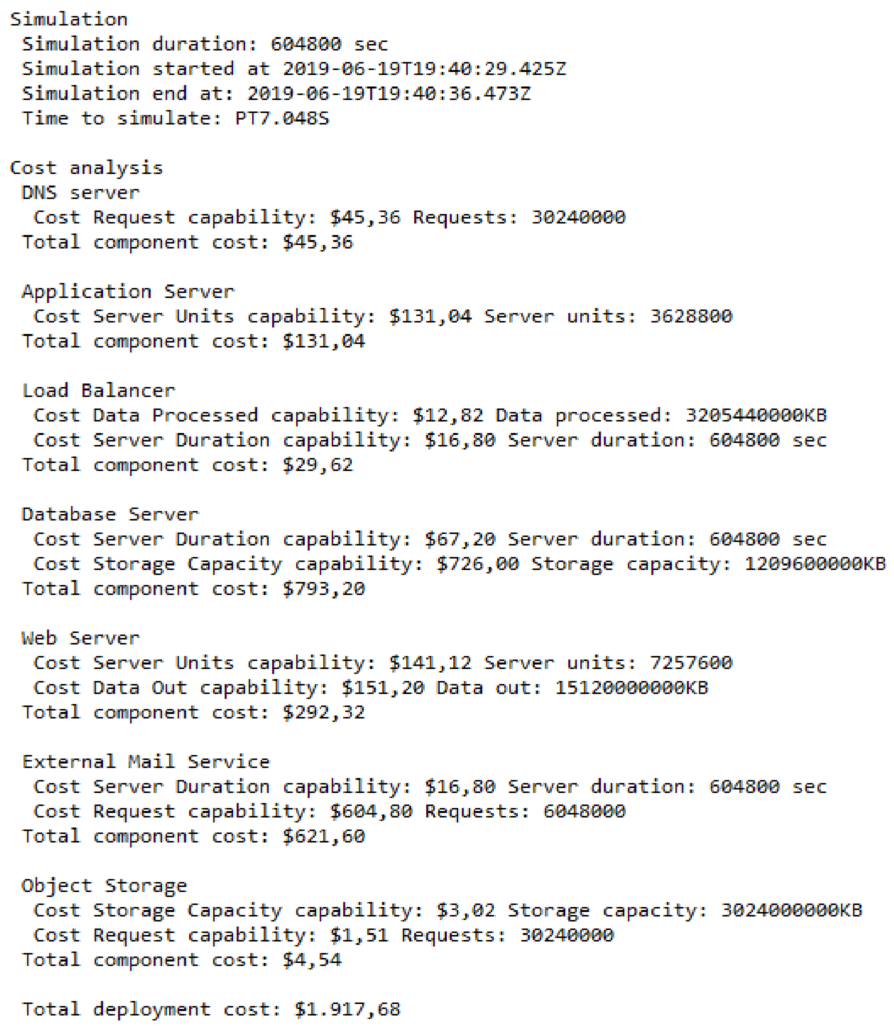


Figure 22 Representation of Traceability between Views

The tool also automatically simulates and calculate the cost of each capability, as well as for each component in the deployment model. In Figure 23, the output of these activities is depicted. The first four rows are specifics about the simulation, like the simulation duration and the process time of the simulation. The rows below are part of the cost analysis activity. For each component the total cost, is given as well as the costs for each associated capability. The last row gives the total costs of the whole simulated deployment.



```
Simulation
 Simulation duration: 604800 sec
 Simulation started at 2019-06-19T19:40:29.425Z
 Simulation end at: 2019-06-19T19:40:36.473Z
 Time to simulate: PT7.048S

Cost analysis
 DNS server
  Cost Request capability: $45,36 Requests: 30240000
 Total component cost: $45,36

 Application Server
  Cost Server Units capability: $131,04 Server units: 3628800
 Total component cost: $131,04

 Load Balancer
  Cost Data Processed capability: $12,82 Data processed: 3205440000KB
  Cost Server Duration capability: $16,80 Server duration: 604800 sec
 Total component cost: $29,62

 Database Server
  Cost Server Duration capability: $67,20 Server duration: 604800 sec
  Cost Storage Capacity capability: $726,00 Storage capacity: 1209600000KB
 Total component cost: $793,20

 Web Server
  Cost Server Units capability: $141,12 Server units: 7257600
  Cost Data Out capability: $151,20 Data out: 15120000000KB
 Total component cost: $292,32

 External Mail Service
  Cost Server Duration capability: $16,80 Server duration: 604800 sec
  Cost Request capability: $604,80 Requests: 6048000
 Total component cost: $621,60

 Object Storage
  Cost Storage Capacity capability: $3,02 Storage capacity: 3024000000KB
  Cost Request capability: $1,51 Requests: 30240000
 Total component cost: $4,54

 Total deployment cost: $1.917,68
```

Figure 23 Output Cost Analysis

The source code of the simulation engine and the EMF meta-model is published on GitHub and is open source (GPL license), which can be used for further research.

9. Evaluation of the solution

How does the proposed solution perform?

In research question 3 we propose a three-step algorithm to determine the workload of a component and in section 8 we created a tool to automatically execute the steps of this algorithm. In this research question, we are evaluating the validity of our algorithm as well as the correctness of our tool. Validity is investigated by comparing our algorithm by another scientific approach to estimate the workload. The correctness is examined by manually executing step 1 and 2 of our proposed three-step algorithm. In a case study with software architects, we also evaluate the user experience of our graphical editor.

9.1. Evaluation method

The aim of our proposed three-step algorithm is to determine the workload of a component, which contains multiple features and is used by several scenarios. The workload of a component is required to determine the server unit capability, which is the sum of active servers for each simulation unit. In research question 2 we proposed three features types and defined that only a process feature impact the workload of a component, therefore in this evaluation only this type of feature is used.

In this evaluation, we are comparing our three-step algorithm with a Petri Model to assess the validity of our estimation. We have chosen to compare our algorithm with a Petri Net model, because such models are also used to estimate the workload of a component (Wells, Christensen, Kristensen, & Mortensen, 2001). This comparison gives a good image how our algorithm performs with another, scientifically proven approach, for estimating workloads. Secondly, we are evaluating our simulation engine with a manual approach to assess the correctness of our simulation engine (e.g. are there any coding errors in our algorithms). For these tests we expect exact the same outcomes, because this implies that we built our simulation engine as intended.

We will start our evaluation with some basic examples to evaluate our proposed solution. Each basic example has an element, part of our solution, like multiple scenarios, multiple components and multiple features in sequence. Therefore, we test for each element separately how our solution performs to Petri Nets and the manual approach. We also evaluate our solution with more complex examples, which combines several concepts in one example. To achieve consistency between models we try to construct the examples on a mechanical matter. Therefore, we introduce pseudo-code for mapping a deployment model to a Petri Net and queueing network model. The pseudo code can be found in Appendix C.

As mentioned in research question 2, we define concepts of minimal required and optimal servers. First concept is straightforward: the minimal number of servers to maintain a stable situation, which is necessary for queueing networks. For the optimal number of servers, we define that the number of servers is sufficient to have a mean queue waiting time less than some threshold (mostly around 150 Ms). Obtaining the mean queue waiting time is straightforward in a queueing network. However, in a Petri net model determining the (mean) queue waiting time is not possible. The reason for this is that a Petri Net model does not have the concept of stability, this means that it can have a utilization greater than 1.0. When this is the case, the transitions will pile up and the waiting time will increase to infinity if the simulation time is also infinite. Normally a simulation will not keep on infinity, but it can be very large (60*60*24*31 = 2.7m. for one month), so the waiting time will also be very large. In a queueing network this will not happen, because then the network is not stable. Therefore, to define stability in a Petri Net we say that stability is reached when each scenario does not have any waiting time at all, after a simulation period of at least 5000 completed events. This implies that we can determine the number of servers without any waiting time. We expect that the determined servers of a Petri net model ranged between the determined minimal and optimal servers of a queueing network.

9.2. Experiments

This section describes the results of the experiments. We start the experiments with simple examples for evaluating basic concepts, like horizontal and vertical behaviour or multiple scenarios. Subsequently, these concepts are merged in increasingly complex examples in order to evaluate our approach, when all concepts act together.

9.2.1. Basic Example with one Feature and one Scenario

The first example consists of one feature and one scenario, which is the bare minimum for a deployment. The service rate of the feature is 10 (e.g. service time is 100ms). The representation of this deployment for all three approaches can be found in Appendix C, section 12.3.3.

For this experiment the arrival rates are 8, 20, 21, 29 and 75 per sec, the results for each arrival rate are listed below in Table 5. For some arrival rates, multiple runs are performed with different server setups. Only if the server setup produces a stable situation, the waiting times can be determined, if this is not the case, it is marked with an **x**.

Table 5 Results Experiment 1

			Petri Net	Manual	Tool	
#	**Arrival Rate**	**Servers**	**Waiting Time**	**Waiting Time (Ms)**	**Waiting Time (Ms)**	**Utilization rate**
1	8	1	0	400	400	0,80
2	8	2	0	19	19	0,40
3	20	2	0	**x**	**x**	1,00
4	20	3	0	44	44	0,67
5	21	3	0	55	55	0,70
6	29	3	0	938	938	0,97
7	29	4	0	43	43	0,73
8	75	8	0	162	161	0,94
9	75	9	0	34	34	0,83

For all approaches, the minimal required servers are similar for the tested arrival rates. Only the edge case when the utilization rate is exactly 1.0, there is a difference. The reason for this is that in queueing theory there is no stable situation when the utilization rate is exactly 1.0; therefore, an additional server is necessary. This explain the discrepancy when the arrival rate is 20 between the Petri Net and the queueing network-based approaches.

When the experiment is corrected for acceptable waiting times, there are more differences regarding the number of servers. There are two factors influencing this difference. First factor is the utilization rate and is amplified by the total number of servers (second factor). The mean queue waiting time increase when the utilization rate also increases. This is seen in run 4, 5 and 6, where the waiting time increases when the utilization rate also increases. The second factor is seen in run 6 and 8, in these runs the utilization rates are both high (>0.90), but the waiting times are much more acceptable for run 8 than for run 6. This is because in run 8 there are 2.5 times more servers compared to run 6. The total number of servers negatively amplifies the impact of a high utilization rate because the workload can be shared on more servers.

9.2.2. Basic Example with two Features and one Scenario

This example is an extension of the first example, but now with two features. Feature A has a service rate of 10 (100ms service time) and Feature B has a service rate of 5 (200ms service time). A representation of such deployment can be found in Appendix C, section 12.3.3. The same set of arrival rates are used from the previous example. The results are in Table 6.

Table 6 Results Experiment 2

			Petri Net	Manual	Tool	
#	**Arrival Rate**	**Servers**	**Waiting Time**	**Waiting Time (Ms)**	**Waiting Time (Ms)**	**Utilization**
1	8	3	0	324	324	0,80
2	8	4	0	54	54	0,60
3	20	6	0	x	x	1,00
4	20	7	0	184	184	0,86
5	21	7	0	309	309	0,90
6	29	9	0	888	891	0,97
7	29	10	0	135	135	0,87
8	29	11	0	48	48	0,79
9	75	23	0	528	531	0,98
10	75	24	0	135	134	0,94
11	75	25	0	61	61	0,90

The results are similar with the results from the previous example. For each arrival rate the minimal required servers are the same, except for the edge case when the utilization is 1.0. When the number of servers is corrected for acceptable mean waiting times, there are more differences between both models.

9.2.3. Basic example with one Feature and two Scenarios

The deployment from the first experiment is used for this example. The feature has a service rate of 10 and instead of one scenario, there are two scenarios using the same feature. This deployment is represented in Appendix C, section 12.3.5.

The same set of arrival rates are used from the previous example. Although the distribution of the arrival rate over two scenarios is different. The results are in Table 7.

Table 7 Results Experiment 3

					Petri Net	Manual	Tool	
#	**Arrival Rate**			**Servers**	**Waiting Time**	**Waiting Time (Ms)**	**Waiting Time (Ms)**	**Utilization**
	Scenario 1	Scenario 2	Total					
1	4	4	8	1	x	400	400	0,80
2	4	4	8	2	0	19	19	0,40
3	10	10	20	2	0	x	x	1,00
4	10	10	20	3	0	44	44	0,67
5	11	10	21	3	0	55	55	0,70
6	14	15	29	3	x	938	938	0,97
7	14	15	29	4	0	43	43	0,73
8	10	19	29	3	0	938	938	0,97
9	23	6	29	3	x	938	938	0,97
10	23	6	29	4	0	43	43	0,73
11	35	40	75	8	0	162	161	0,94
12	35	40	75	9	0	34	34	0,83

An anomaly we found between Petri Net and our solution is the difference in minimal servers for a total arrival rate of 29 (run 6-10). We experienced that the distribution over both scenarios influence the number of minimal required servers. We do not have an explanation for this, but we found out that when an arrival rate of one scenario is divisible without rounding off, the minimal required servers is one unit lower. We expect that it has to do with rounding off the arrival rates in the Petri Net editor[9]. We found no differences between queueing networks and our solution, which is expected.

9.2.4. Complex Example Test Deployment

The basic examples consist of just one additional aspect we proposed in our approach. To increase the validity of our approach we must also test how it perform when all aspects work together. To test this, we use the deployment architecture from research question 2. It consists of two components, nine features and four scenarios. In this deployment almost each aspect, proposed in research question 2, is present. A representation of this deployment for all three approaches can be found in Appendix C, section 12.3.6. The results of the simulation from both models are listed in Table 8 below:

Table 8 Results Experiment 4

			Petri Net	Manual			Tool		
#	**Servers**		**Waiting Time**	**Waiting Time (Ms)**			**Waiting Time (Ms)**		
	Component A	Component B	Total	Total	Component A	Component B	Total	Component A	Component B
1	7	6	X	1062	869	193	1062	869	193
2	8	6	X	296	103	193	296	103	193
3	9	6	X	227	34	193	227	34	193
4	7	7	X	911	869	42	911	869	42
5	8	7	0	145	103	42	145	103	42
6	9	7	0	76	34	42	76	34	42
7	7	8	X	883	869	14	883	869	14
8	8	8	0	117	103	14	117	103	14
9	9	8	0	48	34	14	48	34	14

[9] YASPER http://www.yasper.org/acpn/

The Petri Net model is stable when Component A has at least 8 servers and Component B has at least 7 servers. For both components, this is one server unit off from our solution. When we correct our model for acceptable waiting times, we also get 8 servers for Component A and 7 servers for Component B. This is exact the same amount as the Petri Nets model.

9.2.5. Complex Example E-Voter

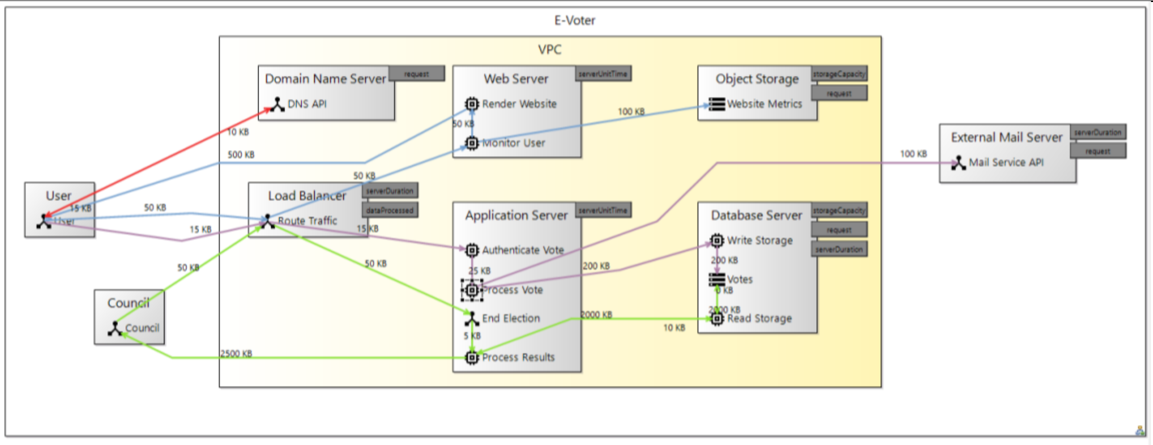
As last example, we are evaluating the E-Voter deployment, which is our running example. The aim of this deployment is to represent a realistic cloud deployment. There are three main scenarios: request website, vote and process results, with arrival rates of 50, 10 and 1 per second. The results of the experiments are in Table 9.

Table 9 Results Experiment 5

				Petri Net	Tool			
#	**Servers**			**Waiting Time**	**Waiting Time (Ms)**			
	Web Server	App Server	Database Server	Total	Total	Web Server	App Server	Database Server
1	10	4	1	X	X	X	1622	X
2	10	4	2	X	X	X	1622	61
3	10	5	1	X	X	X	140	X
4	10	5	2	0	X	X	140	61
5	11	5	2	0	337	136	140	61
6	11	6	2	0	235	136	38	61
7	12	5	2	0	246	45	140	61
8	12	6	2	0	144	45	38	61

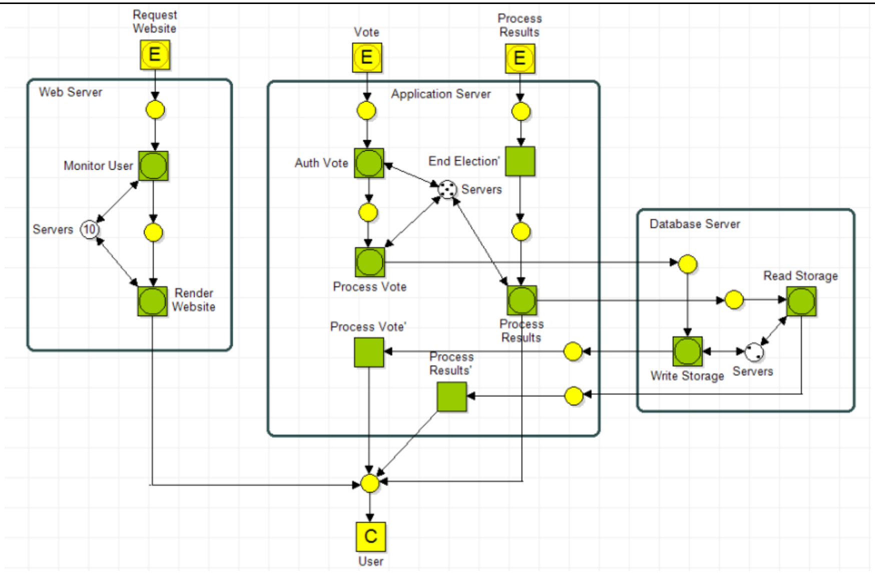
The Petri Net model is stable in the fourth run, while our solution is stable in the fifth run. This is a difference of one server for the Web Server component. In this example the minimal required servers are enough for having acceptable waiting times per component. The models used for this experiment can be found in Table 10.

Table 10 Experiments E-voter example

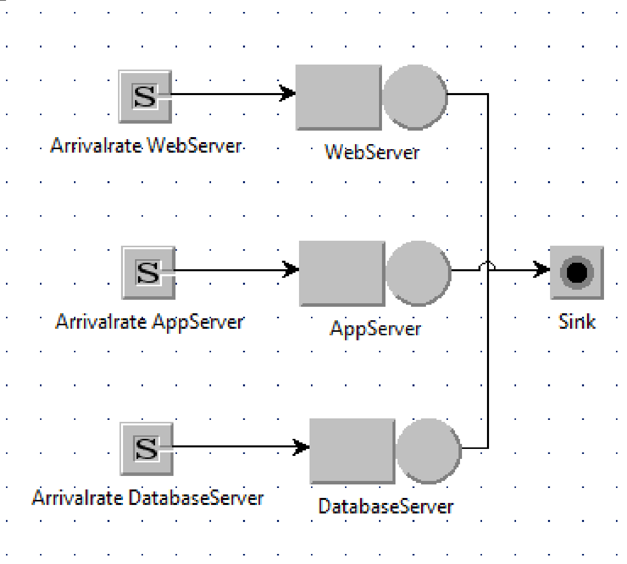


Created tool[10]

[10] This is an old representation of the graphical editor. The code of the simulation engine has not changed, so the experiment is still valid.



Petri Net Model



Manual approach in JMT[11]

9.3. *Case study*

We performed a case study with two software architects at a software company. Software architecture is valuable and highly present at this company, because of their complex software systems. The software architects are enthusiastic about the graphical editor, because of the high-level approach. They stated that, in spite of complex modelling language, the most used language is boxes and lines at their practice. Our solution is also based on boxes and lines, which makes it highly generalizable but also easy to use and understand. This is one the big advantages of using this tool. Another advantage is the persistent storage of our tool. Currently their models are defined on

[11] Java Modelling tools: http://jmt.sourceforge.net

whiteboards, which are easily erasable, which makes it difficult to resume their work and continue at a later moment.

The software architects also identified some drawbacks and opportunities of our solution. They stated that a comparison between different instance types will be valuable, because that is a big part of their daily practices. They also stated that the tool lacks some filter abilities for scenarios, which is preferable because there are many scenarios for their systems. Another identified drawback is the lack of dynamic data transfers over time. This usable to mimic caching etc. The software architects also highlighted that it is useful to export the results of the cost estimations to Excel.

9.4. Conclusion

We tested our solution with a Petri Net model to evaluate the validity. In all five experiments, we find no significant deviations between our solution and the Petri Net model. As expected, the number of required servers for a Petri Net model lies between the required and optimal servers of our algorithm. The results between a manual approach and our created tool are similar. This means that the algorithms in our simulation engine works correctly. In the case study we identified future work which will can be very promising. Altogether the software architects are very enthusiastic about our solution and thinks it is usable in practices. So, we can conclude that our solution fulfils the needs of software architects.

10. Discussion and Conclusion

In this this section the findings of this study are discussed. Subsequently, the validity and limitations are reviewed. Lastly, we propose identified opportunities for further research and a discussion of the conclusion of this study.

10.1. Findings

The aim of this study is to improve the process of defining and structuring a software architecture description and analyse that using operational cost. This is achieved by proposing a step-by-step framework and a tool which support the steps of this framework. The framework is based on our proposed extension of a software architecture meta-model in research question 2. In section 9 we prove that our proposed algorithm of estimating the workload of a component gives similar results as a Petri Net model, which is a scientifically proven method for evaluating workloads. We also interviewed software architects about their experiences with our framework and tool. In this case study we noticed that software architects are enthusiastic to use our proposed solution. This research showed that we improved the process of defining and structuring a software architecture description, but more importantly we incorporate a decision-making perspective based on operational costs.

10.2. Validity

In this section we discuss the validity of the present study.

10.2.1. Construct validity

As this study leans on several meta-models, it is important that these models are constructed carefully. Each meta-model is based on and affirmed by scientific literature and is verified by a second researcher. The algorithms for estimating the workload are based on earlier research by Klock et al. (2017) and are verified by a second researcher as well as evaluated in section 9.

10.2.2. Internal validity

The data collection of research question 1 is discussed sufficiently to reproduce this search. The steps and criteria used in this study are discussed in detail throughout the entire thesis. The tool we created is open source and accessible for everyone to verify or extend the code for further research.

10.2.3. External validity

The proposed framework is composed to be as general and extensible as possible. In principle each kind of software system is supported, as the views are also used in software architecture. In research question 2 we present a mapping between the ISO standard of software architecture and our proposed extension of it. The identified capabilities are proven to be cloud agnostic, because we only select those used by all three cloud providers. Extensibility is achieved by using super types and annotation for certain concepts, which makes it simple to extend such concepts.

10.2.4. Reliability

In the evaluation in section 9 we tested our algorithm with multiple deployments and use cases to identify errors or unexpected behaviour. When we found something, we investigated the cause and explained it extensively. Furthermore, we tested the correctness of our simulation engine in section 9, so that our code works as intended.

10.3. Limitations

Due to time constraints, we only evaluated our framework by a running example and not a real-world software system, although we tried to make the running example as realistic as possible. Testing our framework with a real-world scenario would take at least a month, because some capabilities are only determined at the end of each month. Furthermore, testing our simulation approach with a real-world system requires knowledge about the actual (real world) service rates. This is an entirely new scientific study in itself. Another limitation is that our graphical editor only supports static arrival rates at this moment. This also has to do with time constraints. We expect that incorporating dynamic arrival rates is not difficult, because it is only an extension of the meta-model. The simulation engine already supports dynamic arrival rates, because the arrival rates are evaluated each simulation unit.

10.4. Opportunities

During this research we identified some areas for further research and opportunities:

10.4.1. Extension of capabilities and consumption metrics

In research question 1, we identified seven capabilities and associated metrics for calculating the operational cost. As mentioned earlier, we have opted for these capabilities because they have the most influence on the operational cost, even though more capabilities can be included. This probably means an extension of the meta-model from section 8.1 and adjustments of the simulation engine to support the new capabilities.

10.4.2. Support for more architectural views

At the moment our proposed software architecture description meta-model supports three views; functional, scenario and deployment view. This meta-model can also be extended with the views from the viewpoint catalogue of Rozanski and Woods (2005). The context view, for example, can be used to distinguish whether a component is part of a virtual cloud network or of an external SaaS.

10.4.3. More perspectives for analysing a deployment

The decision-making step in activity 6 of our proposed framework is based on an operational cost perspective. As we discussed in the problem statement, operational cost for cloud deployments is one of the main factors for migrating to a cloud deployment. Although, other perspectives can be added to this activity, the reliability perspective is a good candidate for this. In our meta-model we already take this into account by using annotations for certain architectural elements.

10.4.4. More advanced simulation settings

The software architects stated that a cloud infrastructure is very complex and there are many settings which influence the deployment. To realistically simulate such deployment the simulation must incorporate these settings. One of these settings is a function over time for data transfers, to mimic caching. The ability to model dynamic arrival rates is also essential to realistically simulate a cloud deployment.

10.5. Conclusion

In this section we will answer the research questions specified in section 1.2.

SRQ 1 *What are the constituents of pricing in cloud-based environments?*

In the first research question we investigated scientific literature concerning the price structure of cloud resources. Based on the findings, we proposed a meta-model of a price structure, including a pricing scheme, pricing function and pricing variable. In Table 1, we mapped the concepts used in the field of cloud computing to pricing theory. Based on the meta-model and the mapping of terms we identified seven capabilities (pricing functions) with the most impact on the operational costs.

SRQ 2 How to embed software architecture in a cloud-based environment?

In this research question we used the identified metrics from research question 1 to propose one super meta-model, which incorporate all concepts required to simulate a cloud-based deployment. This meta-model of a software architecture description encompasses three views, the functional, scenario and deployment view. Traceability and extensibility are considered by composing this meta-model. Traceability is achieved by using interelement relationships between features and other architectural elements within all views. For some classes in the meta-model, annotations are specified. This will enhance extensibility. The proposed meta-model is extended with properties in section 8.1 to support the required data for the simulation approach.

SRQ 3 How to simulate an elastic deployment to determine usage metrics?

Based on the seven identified capabilities from research question 1, we formulate a simulation approach to determine the use of these capabilities in a cloud deployment. We found out that most capabilities are simple to obtain, except for the capability *server unit*. The reason for this is that this capability is influenced by the workload of a component, which is difficult to estimate. We propose a three-step algorithm to estimate the workload of component based on multiple features and scenarios. First step is to eliminate horizontal behaviour, with the addition method, second step by eliminating vertical behaviour with the composition method (by Klock et al., (2017)) and as last step determine the required servers based on optimal performance. This is represented in Figure 18. The formalized simulation approach will be used in the (automatic) simulation engine to determine the use of resources and estimate the operational cost of a deployment.

SRQ 4 How does the proposed solution perform?

At last we evaluated the performance of our solution on multiple perspectives. Firstly, we validated our algorithm as well as the correctness of our simulation engine. In comparison with a Petri Net model we get similar results. To evaluate the correctness of our model we compared it with a similar approach, in this comparison we also get the same results as with our simulation, which means that our code works as intended. Secondly, we evaluated the user experience of our framework and tool with software architects. In this case study the results are promising and opportunities for further research are identified.

Research Question: How to improve the process of understanding and analysing the operational costs of an elastic software system based on its architecture.

The main question is answered by our proposed framework and created tool. The framework guides a software architect with a step-by-step approach to define and structure a functional architecture view, which incorporate functional behaviour in a scenario model. In the next steps the deployment view is specified and features (from the functional view) are referenced to components. The behaviour from the scenario view is automatically mapped to the deployment view, because of the traceability between views. At last the pricing function capabilities are added for each component. When this is completed a whole software architecture description is modelled, which is consistent with our proposed meta-model. In the next activity we simulate this model in order to determine the consumption metrics. We created an automatic tool for simulating such deployment, because of the labour-intensive calculations specified in section 6.2. The last activity can be used for different purposes. Main purpose is to analyse the deployment, based on operational cost and assess if the software architecture description is sufficient or not. Another purpose is the ability to compare multiple cloud providers or to audit the invoices of a cloud provider. A representation of this analysing-adjusting cycle can be found in Figure 19.

11. Bibliography

Aboba, B., Arkko, J., & Harrington, D. (2000). Introduction to accounting management. *No. RFC 2975*.

Adan, I., & Resing, J. (2015). *Queueing Systems*. Eindhoven: Eindhoven University of Technology.

Agrawal, A., Jain, V., & Sheikh, M. (2016). Quantitative Estimation of Cost Drivers for Intermediate COCOMO towards Traditional and Cloud Based Software Development. (pp. 85-95). Annual ACM India Conference: ACM.

Allen, A. (2014). *Probability, statistics, and queueing theory*. Cambridge: Academic press.

Al-Roomi, M., Al-Ebrahim, S., Buqrais, S., & Ahmad, I. (2013). Cloud Computing Pricing Models: A Survey. *International Journal of Grid and Distributed Computing*, 93-106.

Alvarez, P., Hernandez, S., Fabra, J., & Ezpeleta, J. (2015). Cost Estimation for the Provisioning of Computing Resources to Execute Bag-of-Tasks Applications in the Amazon Cloud. *Grid Economics and Business Models* (pp. 65-77). Cluj-Napoca: Springer.

Andrikopoulos, V., Song, Z., & Leymann, F. (2013). Supporting the Migration of Applications to the Cloud through a Decision Support System. *6th IEEE International Conference on Cloud* (pp. 565-572). Santa Clara: IEEE Computer Society.

Armbrust, M., Fox, A., Griffith, R., Joseph, A., Katz, R., Konwinski, A., . . . Zaharia, M. (2010). A View of Cloud Computing. *Communications of the ACM*, 50-58.

Barr, J. (2006, 08 25). *Amazon EC2 Beta*. Retrieved from AWS: https://aws.amazon.com/blogs/aws/amazon_ec2_beta/

Bergmayr, A., Troya, J., Neubauer, P., Wimmer, M., & Kappel, G. (2014). UML-based Cloud Application Modeling with Libraries, Profiles, and Templates. *CloudMDE 2014: 2nd International Workshop on Model-Driven Engineering on and for the Cloud co-located with the 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2014)* (pp. 56-65). Valencia: CEUR-WS.

Brinkkemper, S., & Pachidi, S. (2010). Functional Architecture Modeling for the Software. *Software Architecture*, 198-213.

Calheiros, R., Ranjan, R., Beloglazov, A., De Rose, C., & Buyya, R. (2010). CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 23-50.

Chaczko, Z., Mahadevan, V., Aslanzadeh, S., & Mcdermid, C. (2011). Availability and Load Balancing in Cloud Computing. *2011 International Conference on Computer and Software Modeling* (pp. 134-140). Singapore: IACSIT Press.

Choudhary, V. (2007). Comparison of Software Quality Under Perpetual Licensing and Software as a Service. *Journal of Management Information Systems*, 141-165.

Clements, P., Garlan, D., Bass, L., Stafford, J., Nord, R., Ivers, J., & Little, R. (2002). *Documenting software architectures: views and beyond*. London: Pearson Education.

Cloudorado. (2011, June 30). *Cloudorado Blog Is Starting*. Retrieved from Cloudorado: http://blog.cloudorado.com/2011/06/cloudorado-blog-is-starting.html

Coles, C. (2018, July 30). *Cloud Market in 2018 and Predictions for 2021*. Retrieved from Skyhighnetworks: https://www.skyhighnetworks.com/cloud-security-blog/microsoft-azure-closes-iaas-adoption-gap-with-amazon-aws/

Cooper, R., & Kaplan, R. (1991). Profit Priorities from Aotivity-Based Costing. *Harvard business review*, 130-135.

Costello, K. (2019, April 2). *Gartner Forecasts Worldwide Public Cloud Revenue to Grow 17.5 Percent in 2019*. Retrieved from Gartner: https://www.gartner.com/en/newsroom/press-releases/2019-04-02-gartner-forecasts-worldwide-public-cloud-revenue-to-g

Coutinho, E., Sousa, F., Rego, P., Gomes, D., & de Souza, J. (2015). Elasticity in Cloud Computing: A Survey. *Annals of Telecommunications*, 289-309.

Dillon, T., Wu, C., & Chang, E. (2010). Cloud Computing: Issues and Challeng. *The 24th IEEE International Conference on Advanced Information Networking and Applications* (pp. 27-33). Perth: IEEE Computer Society .

Dutta, A., Peng, G., & Choudhary, A. (2013). Risks in Enterprise Cloud Computing: the Perspective of IT Experts . *Journal of Computer Information*, 39-48.

Hevner, A., March, S., Park, J., & Ram, S. (2004). Design science in information systems research. *MIS quarterly 28*, 75-105.

Hosseini, H. (2013, October 14). *How to Model Enterprise Cloud Computing Costs*. Retrieved from Rightscale: https://www.rightscale.com/blog/cloud-cost-analysis/how-model-enterprise-cloud-computing-costs

Ibrahimi, A. (2017). Cloud Computing: Pricing Model. *(IJACSA) International Journal of Advanced Computer Science and Applications*, 434-441.

IDG Cloud Computing. (2018, August 8). *2018 Cloud Computing Survey*. IDG Cloud Computing. Retrieved from IDG: https://www.idg.com/tools-for-marketers/2018-cloud-computing-survey/

International Organization for Standardization. (2011). *ISO/IEC/IEEE 42010:2011(E)*. ISO/IEC/IEEE. Retrieved from https://www.iso.org/standard/50508.html

Islam, S., Lee, K., Fekete, A., & Liu, A. (2012). How a consumer can measure elasticity for cloud platforms. *ACM/SPEC International Conference on Performance Engineering* (pp. 85-96). ACM.

Jallow, A., Hellander, A., & Toor, S. (2017). Cost-aware Application Development and Management using CLOUD-METRIC. *CLOSER* (pp. 515-522). Porto: SciTePress.

Kim, W., Kim, S., Lee, E., & Lee, S. (2009). Adoption Issues for Cloud Computing. *7th International Conference on Advances in Mobile Computing and Multimedia* (pp. 2-5). Kuala Lumpur: ACM.

Klock, S., Van der Werf, J., Guelen, J., & Jansen, S. (2017). Workload-based clustering of coherent feature sets in microservice architectures. *International Conference on Software Architecture (ICSA)* (pp. 11-2-). Gothenburg: IEEE.

Kruchten, P. (1995). Architectural Blueprints—The “4+1” View. *IEEE Software* , 42-50.

Laatikainen, G., Ojala, A., & Mazhelis, O. (2013). Cloud services pricing models. *International Conference of Software Business* (pp. 117-129). Berlin, Heidelberg: Springer.

Mao, M., & Humphrey, M. (2012). A Performance Study on the VM Startup Time in the Cloud. *IEEE Fifth International Conference on Cloud Computing* (pp. 423-430). Honolulu: IEEE.

Medvidovic, N., & Taylor, R. (2010). Software Architecture: Foundations, Theory, and Practice. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, 471-472.

Mell, P., & Grance, T. (2011). *The NIST Definition of Cloud Computing*. Gaithersburg: National Institute of Standards and Technology.

Pahl, C., Xiong, H., & Walshe, R. (2013). A Comparison of On-Premise to Cloud Migration Approaches. *European Conference on Service-Oriented and Cloud Computing* (pp. 212-226). Málaga: Springer.

Patel, C., & Shah, A. (2005). *Cost Model for Planning, Development and Operation of a Data Center*. Palo Alto: Internet Systems and Storage Laboratory.

Patel, J., Jindal, V., Yen, I.-L., Bastani, F., Xu, J., & Garraghan, P. (2015). Workload Estimation for Improving Resource Management Decisions in the Cloud. *IEEE Twelfth International Symposium on Autonomous Decentralized Systems* (pp. 25-32). Taichung: IEEE.

Perry, D., & Wolf, A. (1992). Foundations for the Study of Software Architecture. *Software Engineering Notes*, 40-52.

Pettey, C. (2019, January 29). *Cloud Shift Impacts All IT Markets*. Retrieved from Gartner: https://www.gartner.com/smarterwithgartner/cloud-shift-impacts-all-it-markets/

Rose, C. (1978). A measurement procedure for queueing network models of computer systems. *Computing Surveys (CSUR)*, 263-280.

Rozanski, N., & Woods, E. (2005). *Software systems architecture: working with stakeholders using viewpoints and perspectives*. New York: Addison-Wesley.

Ruiz-Agundez, I., & Garcia Bringas, P. (2011). A Flexible Accounting Model for Cloud Computing. *Annual SRII Global Conference* (pp. 277-284). San Jose CA: IEEE.

Ruiz-Agundez, I., Penya, Y., & Bringas, P. (2012). Cloud Computing Services Accounting. *International Journal of Advanced Computer Research*, 7-17.

Ruiz-Agundez, I., Penya, Y., & Garcia, P. (2010). A Taxonomy of the Future Internet Accounting Process. *The Fourth International Conference on Advanced Engineering Computing and Applications in Sciences* (pp. 111-117). ADVCOMP 2010.

Shepperd, M., & Schofield, C. (1997). Estimating software project effort using analogies. *IEEE Transactions on software engineering*, 736-743.

Soni, A., & Hasan, M. (2017). Pricing schemes in cloud computing: a review. *International Journal of Advanced Computer Research*, 60-70.

Stuckenberg, S., Fielt, E., & Loser, T. (2011). The Impact Of Software-As-A-Service On Business Models Of Leading Software Vendors: Experiences From Three Exploratory Case Studies. *Proceedings of the 15th Pacific Asia Conference on Information Systems (PACIS 2011)* (p. 184). Brisbane: Queensland University of Technology. Retrieved from Webopedia: https://www.webopedia.com/TERM/O/on-premises.html

Tak, B., Urgaonkar, B., & Sivasubramaniam, A. (2011). To Move or Not to Move: The Economics of Cloud Computing. *HotCloud '11*. Portland.

Tang, A., Jin, Y., & Han, J. (2006). A rationale-based architecture model for design traceability and reasoning. *Journal of Systems and Software, 80*, 918-934.

Truonga, H.-L., & Dustdara, S. (2012). Composable cost estimation and monitoring for computational applications in cloud computing environments. *Procedia Computer Science 1*, (pp. 2175-2184). Elsevier.

Van der Ven, J., Jansen, A., Nijhuis, J., & Bosch, J. (2006). Design Decisions: The Bridge between Rationale and Architecture. *Rationale management in software engineering*, 329-348.

Van der Werf, J. M., & Kaats, E. (2015). Discovery of Functional Architectures From Event Logs. *PNSE@ Petri Nets* , 227-243.

Vaquero, L., Rodero-Merino, L., & Buyya, R. (2011). Dynamically Scaling Applications in the Cloud. *SIGCOMM Computer Communication Review*, 45-52.

Walrand, J., & Varaiya, P. (1980). Sojourn times and the overtaking condition in Jacksonian networks. *Advances in Applied Probability*, 1000-1018.

Watts, S., & Hertvik, J. (2018, January 2). *CapEx vs OpEx for IT & Cloud: What's The Difference?* Retrieved from BMC Blogs: https://www.bmc.com/blogs/capex-vs-opex/

Wells, L., Christensen, S., Kristensen, L., & Mortensen, K. (2001). Simulation Based Performance Analysis of Web Servers. *9th International Workshop on Petri Nets and Performance Models* (pp. 59-68). Aachen: IEEE.

Willig, A. (1999). *A Short Introduction to Queueing Theory.* Berlin: Technical University Berlin.

Wood, T., Gerber, A., Prashant, S., Van der Merwe, J., & Ramakrishnan, K. (2009). The Case for Enterprise-Ready Virtual Private Clouds. *Workshop on Hot Topics in Cloud Computing, HotCloud'09*. San Diego: Usenix.

Zhang, M., Ranjan, R., Nepal, S., Menzel, M., & Haller, A. (2012). A Declarative Recommender System for Cloud Infrastructure Services Selection. *Grid Economics and Business Models* (pp. 102-113). Berlin: Springer.

Zseby, T., Zander, S., & Carle, G. (2012). Policy-Based Accounting.

12. Appendices

12.1. Appendix A

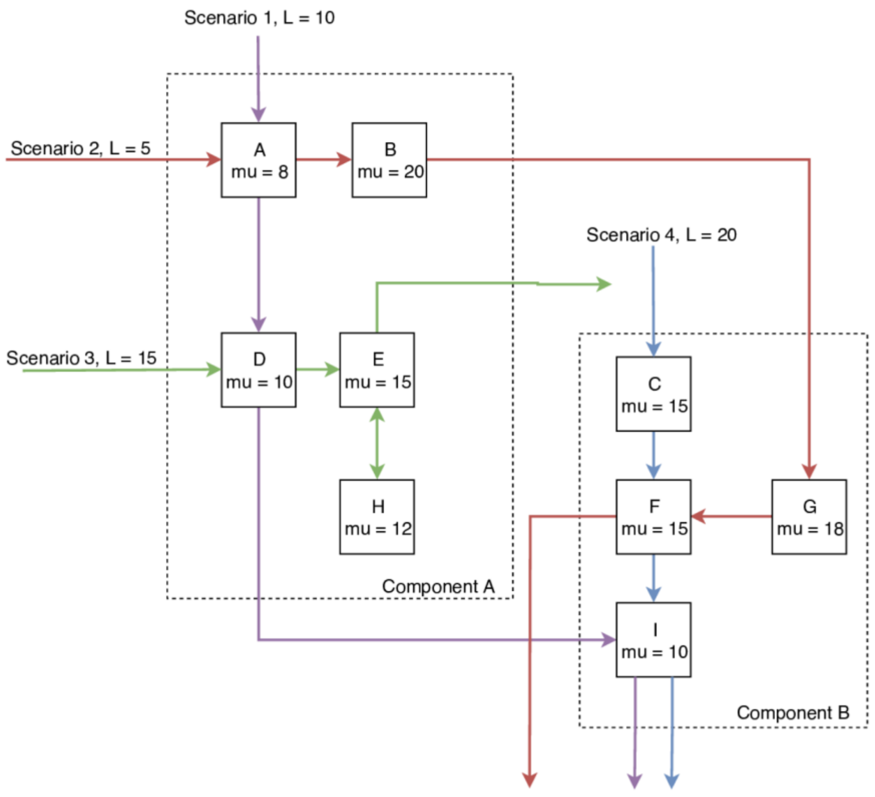
Amazon Web Services			
Resource	Capability	Pricing Function Type	Consumption Metric
Virtual Machine	**Server time**	**Time**	**Seconds**
	Data transfer out	Volume	GB
	Data transfer in	Volume	GB
Block Storage	**Data storage**	**Volume**	**GB**
	Snapshots	Volume	GB
	Throughput	Utilization	IOPS
	Data transfer out	Volume	GB
	Data transfer in	Volume	GB
Load Balancing	**Server time**	**Time**	**Hour**
	Data processed	**Volume**	**GB**
Database	**Server time**	**Time**	**Hours**
	Storage	**Volume**	**GB**
	Backup	Volume	GB
	Data transfer out	Volume	GB
	Data transfer in	Volume	GB
Object Storage	**Data storage**	**Volume**	**GB**
	Request type	**Event**	**Requests**
	Data transfer out	Volume	GB
	Data transfer in	Volume	GB
DNS Server	**Hosted zone**	**Subscription**	**Zone**
	Queries	**Event**	**Requests**

Google Cloud Platform			
Resource	Capability	Pricing Function Type	Consumption Metric
Virtual Machine	**Server time**	**Time**	**Seconds**
	Data transfer out	Volume	GB
	Data transfer in	Volume	GB
Block Storage	**Data storage**	**Volume**	**GB**
	Snapshots	Volume	GB
	Data transfer out	Volume	GB
	Data transfer in	Volume	GB
Load Balancing	**Server Time**	**Time**	**Hours**
	Data processed	**Volume**	**GB**
Database	**Server time**	**Time**	**Hours**
	Storage	**Volume**	**GB**
	Backup	Volume	GB
	Data transfer out	Volume	GB
	Data transfer in	Volume	GB
Object Storage	**Data storage**	**Volume**	**GB**
	Operations	**Event**	**Request**
	Data transfer out	Volume	GB
	Data transfer in	Volume	GB
DNS Server	**Managed zone**	**Subscription**	**Zone**
	Queries	**Event**	**Requests**

Microsoft Azure			
Resource	Capability	Pricing Function Type	Consumption Metric
Virtual Machine	**Server time**	**Time**	**Seconds**
Block Storage	**Data store time** Snapshots **Transaction**	**Time** Volume **Events**	**Months** GB **Transactions**
Load Balancing	**Server time** **Data processed**	**Time** **Volume**	**Hours** **GB**
Database	**Server time** **Storage** Backup	**Time** **Volume** Volume	**Hours** **GB** GB
Object Storage	**Data storage** **Operations**	**Volume** **Event**	**GB** **Requests**
DNS Server	**Hosted zone** **Queries**	**Subscription** **Event**	**Zone** **Requests**
Bandwidth	**Data transfer out** **Data transfer in**	**Volume** **Volume**	**GB** **GB**

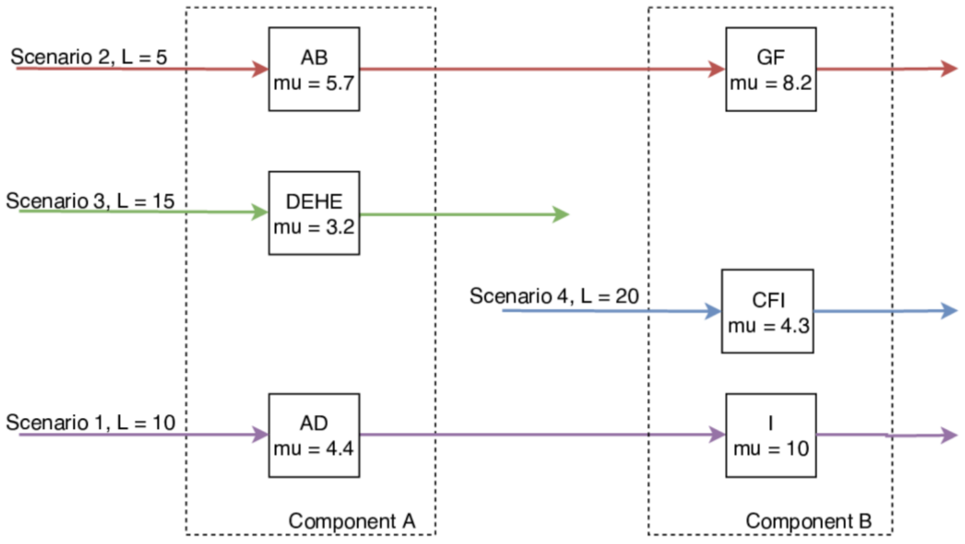
12.2. Appendix B

This is an elaboration of the three-step approach for estimating the workload of a component based on multiple features and scenarios. For this example, a deployment is used with 2 components, 9 features and 4 scenarios (each having a different colour). This deployment is represented in the figure below.



Step 1 – Addition method

In this step all horizontal behaviour is eliminated till only vertical behaviour is left. This is represented in the figure below.



Step 2 – Composition method

In this step all vertical behaviour is merged into one feature, by the method of Klock et al. (2017).

For component A the calculation will be:

$$\text{Lambda merge A} = 5 + 15 + 10 = 30$$

$$\text{Mu merge A} = \frac{5}{30} * 5.7 + \frac{15}{30} * 3.2 + \frac{10}{30} * 4.4 = 4.0$$

For component B the calculation will be:

$$\text{Lambda merge B} = 10 + 5 + 20 = 35$$

$$\text{Mu merge B} = \frac{10}{35} * 10 + \frac{5}{35} * 8.2 + \frac{20}{35} * 4.3 = 6.5$$

Step 3 – Determine servers

In step 3 the newly calculated arrival and service rates are used to determine the required and optimal of number servers in the component.

For component A the calculations will be:

- Minimal required servers = $\left\lceil \frac{30}{4} \right\rceil = 8$ servers Component A
- Optimal servers:
  - 8 servers => waiting time = 403ms
  - 9 servers => waiting time = 85ms

For component B the calculations will be:

- Minimal required servers = $\left\lceil \frac{35}{6.5} \right\rceil = 6$ servers Component B
- Optimal servers:
  - 6 servers => waiting time = 183ms
  - 7 servers => waiting time = 40ms

12.3. Appendix C

12.3.1. Petri Net

- For each Scenario an emitter is modelled
  - **Set** *worktime* = (1000 / service rate) and set *deviation* = 0
- For each Feature invoke a separate transition is modelled
  - IF Feature is used by several Scenarios **set** *name* = “name feature + name scenario’’
  - **Set** *worktime* = (1000 / service rate) and set deviation = 0
- For each Informationflow a case sensitive place is modelled
  - Model *outflow* from the Sender
  - Model *inflow* to the Receiver
- For each component non case sensitive *place* is modelled (
  - **Set** name = Servers
  - IF transition is part of Component a *biflow* between Servers and *transition* (Compute Feature) is modelled
  - IF transition is Connector no biflow is modelled between both[12]
  - **Set** tokens = amount of servers
- A *collector* is modelled as Sink

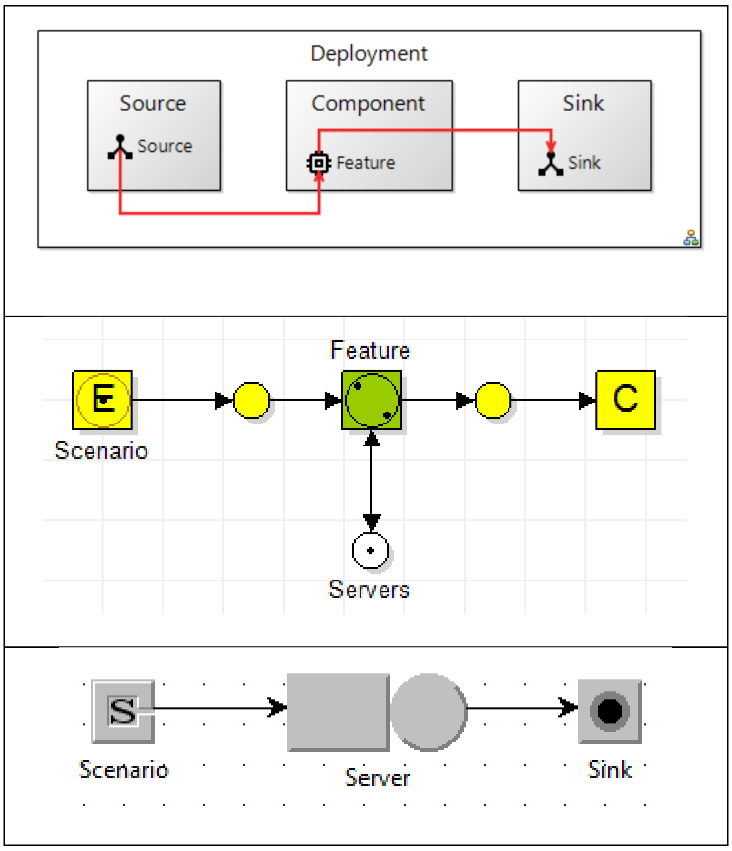
12.3.2. Queueing Network

- For each Scenario a *source* is modelled
- For each Feature invoke a separate class is modelled
  - **Set** *interarrival time distribution* = scenario arrival rate
  - **Set** reference station = associated scenario
- For each Component a *queue* is modelled
  - For each *class* **set** service time distribution = feature service rate
  - IF class is not part of component **set** strategy = zero service time
  - **Set** *number of servers* = amount of servers

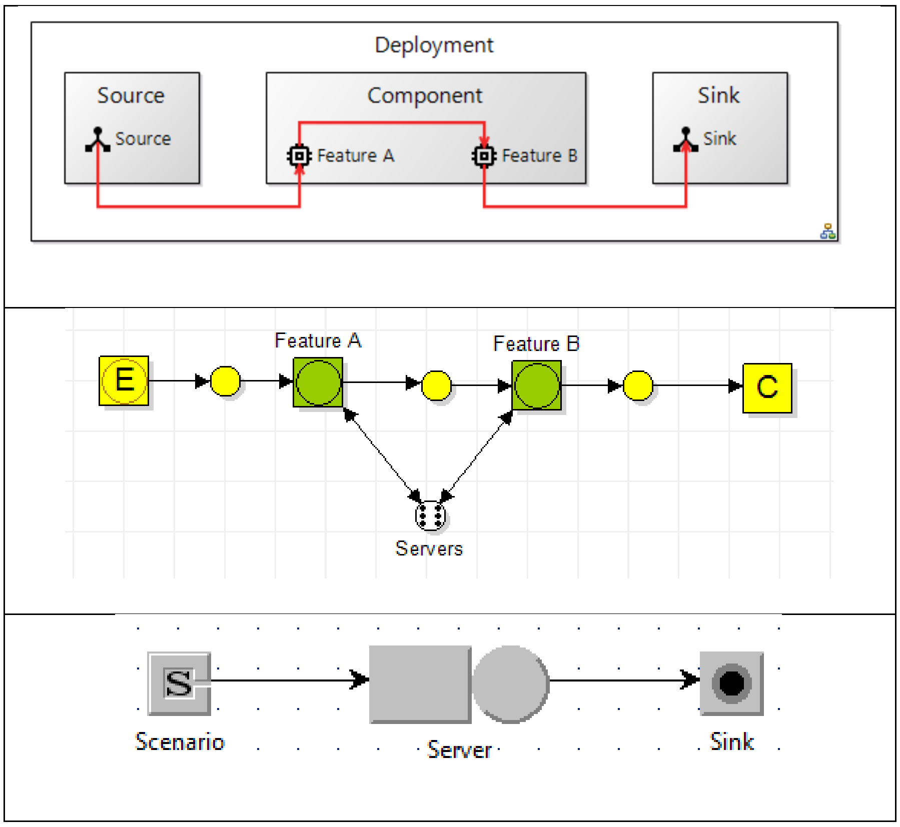
[12] Otherwise it is counted with the workload

- For each connected Component
    - **Set** *connection* between queue
- For each Component model *performance indices*
    - **Set** *queue time*
    - **Set** utilization
- A *sink* is modelled as Sink

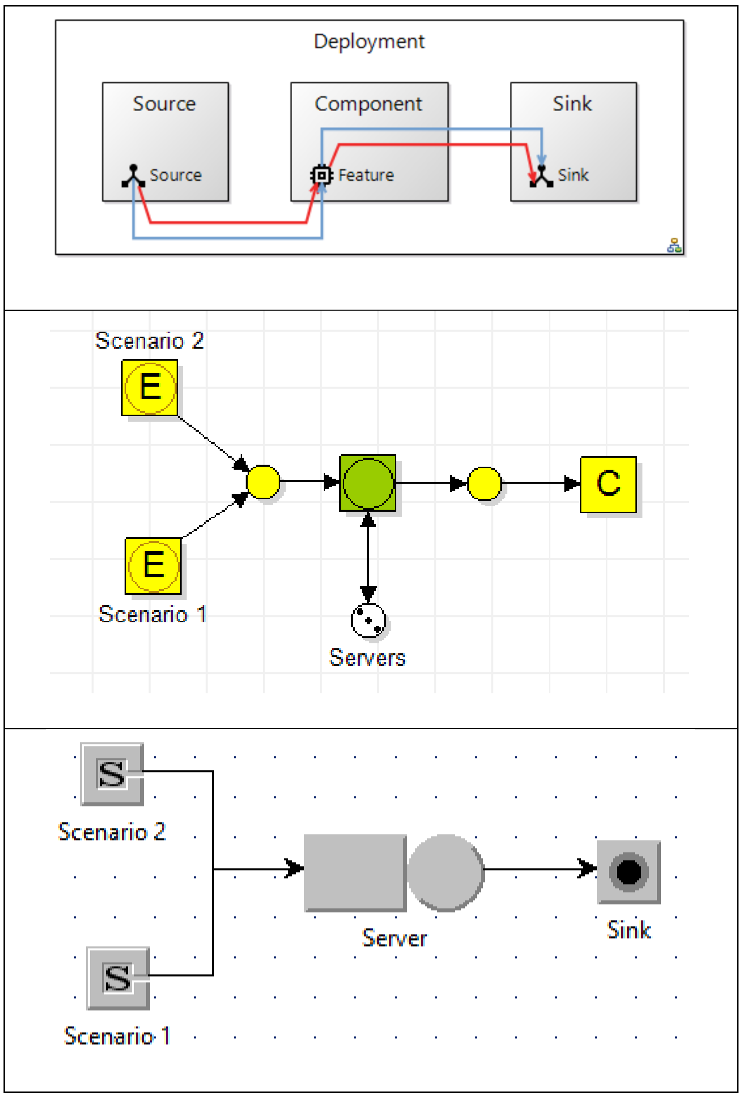
12.3.3. *Basic Example with one Feature and one Scenario*



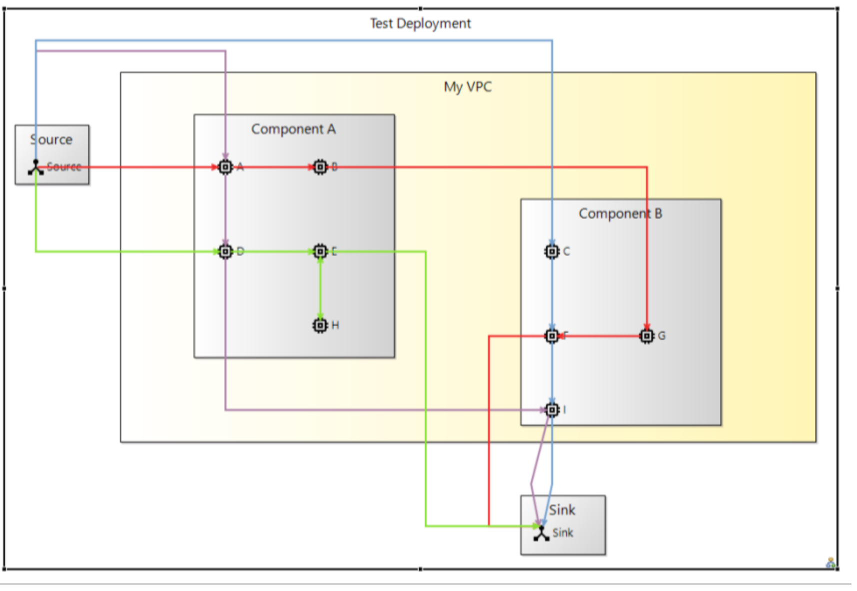
12.3.4. *Basic Example with two Features and one Scenario*

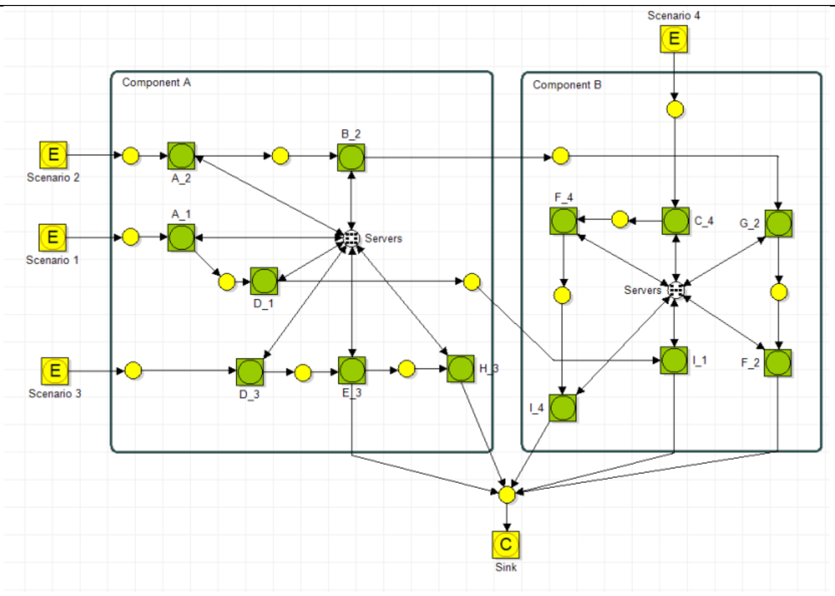


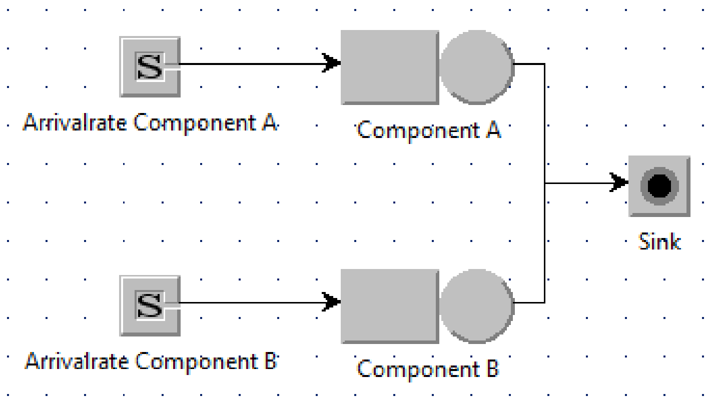
12.3.5. Basic example with one Feature and two Scenarios



12.3.6. Complex Example Test Deployment


Test Deployment
My VPC
Source
Source
Component A
A
B
D
E
H
Component B
C
F
G
I
Sink
Sink


Scenario 4
E
Component A
Component B
E
Scenario 2
A_2
B_2
A_1
Servers
E
Scenario 1
D_1
E
Scenario 3
D_3
E_3
H_3
F_4
C_4
G_2
Servers
I_1
F_2
I_4
C
Sink


S
Arrivalrate Component A
Component A
Sink
S
Arrivalrate Component B
Component B