

Institute of Parallel and Distributed Systems

University of Stuttgart  
Universitätsstraße 38  
D-70569 Stuttgart

Masterarbeit

# **Extending Modeling Concepts of OpenClams to Support Performance Analysis with Layered Queuing**

Simon Matejetz

**Course of Study:** Softwaretechnik

**Examiner:** Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel

**Supervisor:** M.Sc. ETH Inf.-Ing. Otto Bibartiu

**Commenced:** December 8, 2020

**Completed:** June 22, 2021



## Abstract

For the last couple of years, cloud computing has become a more and more prevalent topic in software engineering. The ability to deploy software on a readily provided infrastructure naturally brings many opportunities, especially for smaller businesses. Additionally, many different cloud providers offer so-called backing services. Software developers can use these services as readily available components in their applications instead of implementing their own solutions for recurring use-cases like computation or data storage. The availability of readily available application environments and backing services led to new architectural styles, like microservice and cloud-native. Instead of deploying a monolithic application in a single self-hosted environment, an application can use the advantages of cloud computing much more efficiently if its logical parts are split up into independent components. While splitting up an application into smaller components entails several advantages through the domain of cloud computing, like the possibility to scale each component's environment up or down individually, these new architectural styles inevitably add a new layer of complexity to the system. In a monolithic application, the most critical aspect of its architecture regards the complex internal processes.

In contrast, in a component-based application, each component is kept as simple and small as possible. Therefore, the main complexity stems from the interaction between these independent components instead of their internal behavior. For this reason, software architecture faces new challenges when it comes to the cloud domain when modeling these interactions between cloud components. Because of this, a means to model the components of a cloud application and their interactions after successful initialization is needed. OpenClams is a cloud modeling framework developed at the University of Stuttgart that enables a cloud architect to model a component-based cloud application based on probabilistic user behavior.

Further, OpenClams assists the cloud architect in deciding between different cloud offerings by comparing their cost and availability via so-called evaluation services. An application's architecture, described with Clams, can further be analyzed and optimized for its availability and the predicted costs of the modeled application as a whole. This work further extends the modeling concepts of OpenClams, enabling a cloud architect to attach performance information in the form of annotations to cloud components and their interactions. An additional evaluation service was developed as a part of this thesis which uses these provided pieces of performance information to transform the Clams model into a Layered Queuing Network model to evaluate the approximate utilization of individual components and detect components representing possible bottlenecks in the systems architecture by using standardized tooling for analyzing Layered Queuing Networks.



## Kurzfassung

In den letzten Jahren hat sich Cloud Computing zu einem immer wichtigeren Thema im Bereich der Softwareentwicklung entwickelt. Die Möglichkeit, Software auf einer bereitgestellten Infrastruktur auszubringen, bringt viele Möglichkeiten mit sich, insbesondere für kleinere Unternehmen. Zusätzlich bieten verschiedene Cloud-Anbieter sogenannte Backing-Services an. Softwareentwickler können diese Dienste als fertige Komponenten in ihren Anwendungen nutzen, anstatt eigene Lösungen für wiederkehrende Anwendungsfälle wie Computing oder Datenspeicherung zu implementieren. Diese Verfügbarkeit von fertigen Anwendungsumgebungen und Backing Services führte zu neuen Architekturstilen, wie Microservice und Cloud-Native. Anstatt eine monolithische Anwendung in einer einzigen selbst bereitgestellten Umgebung zu betreiben, kann eine Anwendung die Vorteile des Cloud Computing viel effizienter nutzen, wenn ihre logischen Bestandteile in unabhängige Komponenten aufgeteilt werden. Auch wenn die Aufteilung einer Anwendung in kleinere Komponenten durch die Domäne des Cloud Computings mehrere Vorteile mit sich bringt, wie z. B. die Möglichkeit, jede Komponente individuell zu skalieren, fügen diese neuen Architekturstile dem System unweigerlich eine neue Ebene der Komplexität hinzu. Bei einer monolithischen Anwendung betrifft der wichtigste Aspekt ihrer Architektur die komplexen internen Prozesse. Im Gegensatz dazu wird bei einer komponentenbasierten Anwendung jede Komponente so simpel und klein wie möglich gehalten. Die Hauptkomplexität ergibt sich daher nicht mehr aus dem internen Veralten ihrer Komponenten sondern aus der Interaktion zwischen diesen unabhängigen Bestandteilen. Aus diesem Grund steht die Softwarearchitektur in der Cloud-Domäne vor neuen Herausforderungen, wenn es um die Modellierung dieser Interaktionen zwischen Cloud-Komponenten geht. Daher benötigen wir ein Mittel, um die Komponenten einer Cloud-Anwendung und ihre Interaktionen nach erfolgreicher Initialisierung zu modellieren. OpenClams ist ein an der Universität Stuttgart entwickeltes Cloud-Modeling-Framework, das es einem Cloud-Architekten ermöglicht, eine komponentenbasierte Cloud-Anwendung auf der Basis von probabilistischem Benutzerverhalten zu modellieren.

Weiterhin unterstützt OpenClams den Cloud-Architekten bei der Entscheidung zwischen verschiedenen Cloud-Angeboten, indem es deren Kosten und Verfügbarkeiten über sogenannte Evaluation Services vergleicht. Die mit Clams beschriebene Architektur einer Anwendung kann darüber hinaus hinsichtlich ihrer Verfügbarkeit und der prognostizierten Kosten der modellierten Anwendung als Ganzes analysiert und optimiert werden. Diese Arbeit erweitert die Modellierungskonzepte von OpenClams weiter und ermöglicht es einem Cloud-Architekten, Cloud-Komponenten und deren Interaktionen mit Performance-Informationen in Form von Annotationen zu versehen. Im Rahmen dieser Arbeit wurde ein zusätzlicher Evaluation Service entwickelt, der diese bereitgestellten Performance-Informationen nutzt, um das Clams-Modell in ein Layered-Queuing-Network-Modell zu transformieren, um anschließend die ungefähre Auslastung der einzelnen Komponenten herauszufinden und Komponenten zu erkennen, die mögliche Engpässe in der Systemarchitektur darstellen, indem standardisierte Werkzeuge zur Analyse von Layered-Queuing-Networks verwendet werden.



# Contents

<b>1</b>	<b>Introduction</b>	<b>19</b>
1.1	Problem Statement . . . . .	20
1.2	Approach . . . . .	20
1.3	Thesis Structure . . . . .	20
<b>2</b>	<b>Related Work</b>	<b>23</b>
<b>3</b>	<b>System Model</b>	<b>25</b>
3.1	Cloud Application . . . . .	25
3.2	Components . . . . .	26
3.3	Communication . . . . .	26
<b>4</b>	<b>Fundamentals</b>	<b>29</b>
4.1	Cloud Computing . . . . .	29
4.2	Clams . . . . .	29
4.3	Layered Queuing Network . . . . .	33
<b>5</b>	<b>Concept &amp; Implementation</b>	<b>37</b>
5.1	Concept . . . . .	37
5.2	Implementation . . . . .	45
5.3	Solving the Layered Queuing Network model . . . . .	49
<b>6</b>	<b>Case Study</b>	<b>55</b>
6.1	Model . . . . .	55
6.2	Execution . . . . .	57
<b>7</b>	<b>Conclusion and Outlook</b>	<b>63</b>
	<b>Bibliography</b>	<b>65</b>





## List of Figures

4.1	Visual representation of an Sequence Diagram (SQD) in the OpenClams web interface depicting the interaction between instances of cloud components for the “Login”-State . . . . .	31
4.2	Visual representation of a User Profile in the OpenClams web interface depicting the interaction of a user with a simplified webshop application . . . . .	32
4.3	Cloud Application Modeling Solution (Clams) component tree for the <i>Stateful Component</i> pattern and its refinement options . . . . .	34
4.4	Layered Queuing Network (LQN) with processors, tasks and entries . . . . .	35
5.1	LQN workload layer representing the User Profile (UP) from Figure 4.2 created by following the transition rules from Table 5.1 . . . . .	40
5.2	LQN service layer representing the SQD from Figure 4.1 created by following the transition rules from Table 5.3 . . . . .	43
5.3	<i>Registration</i> SQD using the same <i>User DB</i> instance as the <i>Login</i> SQD from Figure 4.1	46
5.4	Resulting LQN service layer when transforming a Clams model using the same <i>User DB</i> instance in multiple SQDs . . . . .	46
5.5	The avgConcurrency meta attribute added to the <i>User Database (DB)</i> instance . . . . .	47
5.6	Message meta attributes of a synchronous request between two instances . . . . .	47
5.7	Additional “Predict Performance” option in the dropdown menu for selecting evaluation services . . . . .	47
5.8	Dialog asking for the workload attributes for an invocation of the performance evaluation service . . . . .	48
6.1	UP modeling the user interaction with in a webshop application . . . . .	56
6.2	LQN model transformed from Clams model with shortened depiction of the service layer containing only the service tasks for the <i>User Service</i> and <i>User DB</i> instances	57



## List of Tables

5.1	Transformation of Clams UP components to LQN components and their visual representation . . . . .	39
5.2	Workload attributes to be provided additionally to the Clams model . . . . .	42
5.3	Transformation of SQD components to LQN components . . . . .	43
5.4	Performance annotations of instances and messages in a Clams SQD . . . . .	44



## List of Listings

5.1	Annotated Clams model serialized as Javascript Object Notation (JSON) . . . . .	52
5.2	LQN processors and tasks as represented in the Layered Queuing Network Solver (LQNS) input format . . . . .	53
5.3	LQN entries as represented in the LQNS input format . . . . .	54
5.4	LQN activities as represented in the LQNS input format . . . . .	54
5.5	Mean delay per rendezvous as noted in the LQNS result format . . . . .	54
5.6	Utilization per task and entry as represented in the LQNS result format . . . . .	54
6.1	Result of performance analysis with mean delay of 1000 milliseconds . . . . .	58
6.2	Result of performance analysis with average arrival rate of 500 milliseconds . . . . .	59
6.3	Result of performance analysis with average arrival rate of 250 milliseconds . . . . .	59
6.4	Result of performance analysis with average arrival rate of 100 milliseconds . . . . .	59
6.5	Result of performance analysis with average arrival rate of 50 milliseconds . . . . .	59
6.6	Result of performance analysis with average arrival rate of 30 milliseconds . . . . .	60
6.7	Result of performance analysis with average arrival rate of 30 milliseconds and <i>User DBs</i> “maxConcurrency” annotation increased to 100 . . . . .	60
6.8	Result of performance analysis with average mean arrival rate of 5 milliseconds and all stateful components “maxConcurrency” annotation at 100 . . . . .	61
6.9	Result of performance analysis with average arrival rate of 3 milliseconds and all stateful components “maxConcurrency” annotation at 100 . . . . .	61



## List of Algorithms

5.1	Removing backlinks and redistributing branch weights starting from the dot state	41
5.2	Creating and connecting tasks, entries for multiple SQDs . . . . .	45





# Acronyms

- ACID** Atomicity, Consistency, Isolation, Durability. 20
- AWS** Amazon Web Services. 33
- BNF** Backus-Naur form. 49
- CBSE** Component-Based Software Engineering. 19
- Clams** Cloud Application Modeling Solution. 9, 19
- ClamsML** Clams Modeling Language. 29
- CML** Cloud Modeling Language. 20
- CPSM** Cloud-Provider Specific Model. 30
- CPU** Central processing unit. 34
- CSP** Cloud Service Provider. 26
- DB** Database. 9, 20
- DBaaS** Database as a Service. 26
- GUI** Graphical User Interface. 46
- HTTP** Hypertext Transfer Protocol. 48
- IaaS** Infrastructure as a Service. 19
- IoT** Internet of things. 57
- IPVS** Institute for Parallel and Distributed Systems. 19
- IT** Information Technology. 19
- JSON** Javascript Object Notation. 13, 45
- LQN** Layered Queuing Network. 9, 20
- LQNS** Layered Queuing Network Solver. 13, 36
- LTS** Labeled Transition System. 23
- MDA** Model-Driven Architecture. 30
- MSC** Message Sequence Charts. 23
- NIST** National Institute of Standards and Technology. 29
- npm** Node Package Manager. 48

<b>OS</b>	Operating System.	29
<b>PaaS</b>	Platform as a Service.	19
<b>QoS</b>	Quality of Service.	19
<b>RPC</b>	Remote Procedure Call.	26
<b>S2P</b>	Scenario to Performance.	23
<b>SaaS</b>	Software as a Service.	19
<b>SAME</b>	Systems Architecture and Model Extraction.	23
<b>SOA</b>	Service-Oriented Architecture.	19
<b>SPE</b>	Software Performance Engineering.	20
<b>SQD</b>	Sequence Diagram.	9, 23
<b>UCM</b>	Use Case Map.	23
<b>UML</b>	Unified Modeling Language.	21
<b>UP</b>	User Profile.	9, 23
<b>VM</b>	Virtual Machine.	29

# 1 Introduction

In the recent years, cloud computing has become a more and more vital subject in computer science and software engineering [Ley09]. Cloud computing offers readily available infrastructure in a pay-per-use manner. Only paying for the resources needed at any given time means a great advantage for software providers. They don't need to buy and maintain physical servers of fixed processing power anymore and instead leverage cloud resources. These cloud resources come in the form of different service offerings, namely Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). Component-Based Software Engineering (CBSE) has already been a subject of computer science in the last century. [BW98] describes the potential advantages that a component-based software architecture promises, like the reusability of said components. The CBSE approach was taken further by the domain of Service-Oriented Architecture (SOA), which not only describes the modeling of reusable software components but stand-alone services following an always-on semantic [Ars04]. The emerging field of cloud computing with its new provisioning models led to the rise of new architectural styles following the CBSE and SOA principles. One of these new architecture styles is microservice architecture. It is especially well-suited for the provisioning in the cloud as its minimalistic, component-based structure allows to take the greatest advantages of cloud computing, especially regarding the PaaS and SaaS service models. For example, provisioning an application that consists of multiple independent services instead of a single monolith allows to individually scale each service as needed for a given workload [NMMA16]. Microservices that only interact via their interfaces also provide more flexibility. Each service can be modified independently without influencing others as long as its endpoints do not change, which provides great flexibility—for example, reimplementing any service in a different programming language, also giving independence to the developing teams. For these reasons, many big players in Information Technology (IT), like Amazon and Netflix, are following the microservice architecture in the development of their applications [NMMA16].

To assist cloud architects in designing such a microservice-based cloud application that follows the best practices of cloud computing, Cloud Application Modeling Solution (Clams), a tool-supported cloud modeling language, has been developed at the Institute for Parallel and Distributed Systems (IPVS) faculty<sup>1</sup> of the University of Stuttgart. Using Clams, the architect of a cloud application can leverage the structural cloud computing patterns proposed in [FLR+14] to abstractly model cloud components and their interactions that are necessary to implement previously defined use cases. The abstract components can later be refined into concrete cloud offerings. For the refinement step, Clams contains so-called evaluation services that evaluate a Clams model for various Quality of Service (QoS), like cost and availability, on a component and application level to enable well-informed decision making when designing a cloud application.

---

<sup>1</sup><https://www.ipvs.uni-stuttgart.de/>

### 1.1 Problem Statement

As described, designing an application following the microservice architecture has several advantages from a performance perspective, especially if the application uses cloud infrastructure. It is best practice to split an application's microservices into two groups - stateful and stateless. This differentiation is made because there is a difference in scaling microservices of the two groups. Because an arbitrary number of instances of a stateless microservice can run in parallel, it is easy to scale them up or down by adding more or taking away service instances according to workload.

The same does not apply to stateful microservices, especially if they follow the Atomicity, Consistency, Isolation, Durability (ACID) properties. Because state kept in a stateful microservice needs to be consistent and highly available it can not tolerate partitioning following the CAP theorem. Therefore, a stateful microservice following the ACID properties can only run on a single instance. Additionally to only being vertically scalable, most stateful resources, like Databases (DBs), enforce boundaries on the number of concurrent connections to guarantee the ACID properties.

This means that even if the majority of microservices in an application is stateless and therefore scalable almost indefinitely in a cloud environment, a single stateful microservice reaching its performance limits because of the rising number of requests received by the growing number of stateless microservices can propagate the resulting delay up the call chain. Once this delay becomes large enough, it results in a perceived slow down or breaking of the application by the user and thus ruins the user experience. Detecting such a performance bottleneck as early as possible in the lifespan of an application, ideally at design time, is the stated goal of Software Performance Engineering (SPE) and of great importance as the costs to fix the bottleneck grow over an applications lifetime [WFP07].

### 1.2 Approach

Clams provides evaluation services to analyze the cost and availability of cloud applications modeled with its Cloud Modeling Language (CML). This work extends the Clams ecosystem by further extending its modeling concepts to allow the cloud architect to provide performance annotations to interactions between components to enable a performance analysis of the model following the SPE principles. More precisely, an additional evaluation service for the Clams reference implementation OpenClams has been developed that assists the cloud architect in detecting threats of possible bottlenecks in a Clams model that arise from stateful microservice components with limited scalability. For this, the Clams model is transformed into an Layered Queuing Network (LQN) performance model.

The LQN is an extended queuing model that has been proposed for analyzing the utilization and throughput of software systems composed of elements that interact in multiple layers. Standardized tooling to solve LQNs has been proposed by Franks et al., [FMW+05] that will be used to analyze the generated LQN for its performance attributes.

## 1.3 Thesis Structure

This thesis is structured as follows:

**Chapter 2 - Related Work** features scientific work that also concern LQNs in a way that is interesting for the following parts of the thesis, e.g. proposals for the transformation of various Unified Modeling Language (UML) models into LQN models

**Chapter 3 - System Model** describes the features an application needs to possess in order to be used with Clams and more specifically the means developed in this work

**Chapter 4 - Fundamentals** gives an insight into the subjects this work is based on, more specifically cloud computing, Clams and LQN.

**Chapter 5 - Concept & Implementation** presents the concept and implementation of the main contribution of this thesis in the form of enabling performance analysis for Clams models

**Chapter 6 - Case Study** contains a case study that has been carried out using the Clams model of a web shop cloud application including performance annotations to analyze its components for bottlenecks and their utilization

**Chapter 7 - Conclusion and Outlook** concludes the results of this thesis and gives an outlook into possible future work and research that arises from this thesis



## 2 Related Work

This work extends Clams analytical capabilities by transforming extended Clams models into LQN performance models. The Sequence Diagram (SQD) and User Profile (UP) diagrams of the Clams model are based on the Message Sequence Charts (MSC) and Labeled Transition System (LTS) diagrams proposed by Uchitel et al. [UKM03], [UKM04] that share similar elements and structures with UMLs SQD and activity diagram.

Proposals for the transformation of various UML models into LQN models have been brought forward in the last couple of years. In the following, the concepts of these and other approaches that this work was inspired by are described.

In 2001, Israr [Isr01] proposed a lightweight model building technique, called Systems Architecture and Model Extraction (SAME), in order to turn communication traces of distributed systems components, in the form of timestamped *send* and *receive* logs, into communication trees and finally into an LQN model to analyze the traced systems performance. The works' main focus is on detecting communication types and the generation of communication trees from the available system traces. This is not interesting when it comes to Clams models, as not only independent traces at each distributed component but full information about which components interact with each other is given in the Clams SQDs. However, the algorithm to build the LQN model from communication between components, that has been proposed by Israr, inspired the solution developed in this thesis and will be explained more thoroughly in Chapter 5.

Petriu and Woodside [PW05] came forward with the Scenario to Performance (S2P) algorithm to transform Use Case Maps (UCMs) into LQN models. UCMs are structurally very similar to Clams UPs as they also model user behavior regarding a system with the main difference being that the UP does not support OR forks/joins or AND forks/joins and instead works with transition probabilities. Regardless, the representation of the user flow as an activity graph in the LQN models entries as proposed by Petriu and Woodside has also been adapted to model the complex state-based user workload for the LQN model in the solution presented in this thesis.

For the Palladio Component Model, Koziolok and Reussner [KR08] have proposed a similar approach, turning the annotated control flow and interactions of Palladio Components into an LQN model. The main difference to our approach is that, instead of software components of arbitrary size, in this thesis the application to be analyzed is expected to consist of microservices - each running in independent cloud environments. This work also does not focus on creating reusable results for each component but instead on identifying bottlenecks that arise from single components in the analyzed application as a whole.





## 3 System Model

This chapter presents the conditions an application needs to fulfill in order to be analyzed using the approach presented in this thesis. The proposed solution extends the Clams ecosystem; therefore, only applications that can be described via Clams are considered. Further, an application is expected to fulfill certain structural and behavioral requirements defined in the following sections to enable performance analysis and bottleneck detection.

### 3.1 Cloud Application

The system is expected to be a microservice-based application that has been optimized for cloud usage, called a cloud application. The system model of this thesis follows the IDEAL properties, proposed in [FLR+14], as described in the following.

#### IDEAL

The IDEAL properties as proposed by Fehling et al., [FLR+14] contain five features that a cloud application should possess in order to make optimal use of cloud infrastructure. The following listing explains the role of each of the IDEAL properties in the system model.

Isolated State: All state, e.g. application and session state, must be kept in a dedicated, minimal number of components to keep all the other components stateless. An example of such a stateful component is a service with only the responsibility to keep user information in a relational database.

Distribution: The application is expected to consist of multiple components, ideally following a micro-service architecture with each of the components having exactly one well-defined and fine-grained responsibility [NMMA16].

Elasticity: Computation resources can be added and removed during the runtime as needed to, e.g., adapt to the user workload. This property effectively enables almost unbounded horizontal scaling of stateless components as it is possible to add new instances at any time when needed.

Automated Management: The management of a deployed application is automated, e.g., adding more instances as the workload increases. The automated management aspect is not of interest for this work, as application deployment and management are not considered, and is only mentioned at this point for reasons of completeness.

Loose Coupling: Dependencies among application components are minimal as they interact in a message-oriented manner over their interfaces via asynchronous and synchronous communication.

### 3.2 Components

The application components are expected to be dedicated microservices running in environments independently from other services and kept as simple and small as possible, ideally only having a single responsibility in the system. As mentioned in the previous section, handling state is a crucial factor when designing a cloud application. This work assumes that dedicated components contain all kind of state existing in the application and that, ideally, at least one such stateful component exists. That is the case because the main focus of this work is to detect bottlenecks and performance-critical components in modeled system architectures. If all components of a system architecture were to be stateless, practical unlimited horizontal scaling could be achieved due to the elastic nature of the cloud.

Although it is possible to conduct a performance analysis on a fully stateless application using the solution proposed in this thesis, it would limit the result to only finding the average number of active instances/requests on any given workload for each component. Due to the unlimited horizontal scalability, the presence and thus detection of a bottleneck would not be possible.

Further, the maximum number of concurrent connections supported by each stateful component needs to be known or reasonably estimated. For instance, if the stateful component is an instance of a Database as a Service (DBaaS) offering, the connection limits for a service plan can typically be found in the documentation of the Cloud Service Provider (CSP)[Goo][Mic].

### 3.3 Communication

The following section specifies the information that must be known about the communication between components in an applications model to analyze its performance as proposed in this work. When referring to communication between cloud components in this work, exchanging messages in a point-to-point manner is meant. Systems that include broadcast communication, like pubsub, can not be modeled with Clams and are therefore not considered in this thesis. Further technical details of such a point-to-point communication do not need to be defined. Because of this, the communication between two components can be implemented with the means of any technology, like Remote Procedure Call (RPC) or messaging in a synchronous or asynchronous manner.

Assuming there are two components called component A and component B, a synchronous call from component A to component B implies that component A is blocked until the reply from component B is received. In this context, blocking means that the specific task of component A, that sent out the request, can not process further requests from any source until it receives a reply from component B. Component A might still be able to process additional requests as making a synchronous call blocks only one of its worker threads and cloud components are usually heavily parallelized. Therefore, the complete blocking of a component only happens when all available worker threads are blocked at the same time. This is what is considered a bottleneck in this work and can only appear with stateful components because they are not horizontally scalable.

In contrast, an asynchronous message does not block the requesting component A and instead acts as a simple call to component B. The worker thread of component A sending out the request stays free and ready to send out or receive further requests immediately after.

The source and target components need to be specified in the application model for each communication between two components. Additionally, the average service time that a request imposes

on the target component must be specified in the architecture. It also needs to be explicitly stated if communication is of synchronous or asynchronous nature and in which order the interactions appear.



## 4 Fundamentals

### 4.1 Cloud Computing

The National Institute of Standards and Technology (NIST) defines *cloud computing* as a computing model that enables ubiquitous, convenient, and on-demand network access to a shared pool of configurable computing resources. Such computing resources can range from networks, servers, and storage to provided applications and services. These resources can be rapidly provisioned and released with only a little management effort [MG+11]. The NIST further defines three different service models, namely IaaS, PaaS, and SaaS, with the amount of infrastructure that is getting outsourced increasing with each service model as IaaS describes only the usage of a third parties hardware infrastructure, for example, in the form of a Virtual Machine (VM) without any preinstalled software that goes beyond an Operating System (OS). In the PaaS model, the whole environment that an application requires, like compilers, interpreters, and runtimes, is readily provided so that the application only needs to be deployed on the PaaS cloud offering to be functional. The third and last type of service model, SaaS, describes the usage of a software application that is already up and running and entirely managed by the CSP. Such a provided service reaches from complete end-user software to so-called backing services that can be consumed by other applications, e.g., a ready-to-use database, called a DBaaS, or a messaging service [MG+11].

### 4.2 Clams

In agile software development, most commonly, one of the first artifacts created are user stories. A user story is a description of how a user will be interacting with the resulting software system. Clams, therefore, bases on the idea of modeling cloud applications depending on so-called UPs that are introduced more precisely in the following Section 4.2.2. A UP represents each action a user can take in the modeled system as a state, which links to an SQD that models the interaction between system components necessary to enable the action taken by the user. To assist cloud architects in deriving a cloud application architecture from user stories, Clams provides the Clams Modeling Language (ClamsML), a modeling language containing UPs, SQDs, and their components. Also, web-based tooling for the visual representation of a Clams model and the mutation and inspection of the model exists as an open-source project named OpenClams<sup>1</sup>. Clams has similarities with other component-based CMLs like Blueprint [NLT+11] and CAML [BBK+16],[BTN+14], that leverage UML diagrams to model cloud applications orchestration and deployment, but additionally provides the possibility of refining the components via a component tree as is described in Section 4.2.2. To avoid forcing the cloud architect to decide on specific service offerings at an early stage in the

---

<sup>1</sup><https://github.com/openclams>

design time of an application, Clams leverages the structural cloud computing patterns presented in [FLR+14] as abstract modeling components. Using these cloud pattern components in a model enables modeling cloud applications on a conceptual level even before the cloud architect decides on particular cloud offerings for the application's components. Therefore, using Clams, it is possible to model cloud applications in a cloud offering-agnostic way and then refine the modeled components - fluidly transforming the model. This fluid transformation is a new approach to refining a cloud architecture model. Other proposals, like the MODAClouds project brought forward by Di Nitto et al. [DMPS17], follow a Model-Driven Architecture (MDA) approach for creating cloud architectures. They also enable the design of cloud offering-agnostic models and their later refinement into Cloud-Provider Specific Model (CPSM), but do so in a stage-based manner by transforming entire application models instead of individual components as is done in the Clams approach. Leveraging cloud patterns and specific cloud offerings, cloud offering-agnostic, as well as cloud offering-specific components, can be present in a Clams model at the same time.

Clams further assists the cloud architect in decision making when refining a cloud offering-agnostic component into a specific cloud offering by providing insight into offerings that are compatible children of the current component in the component tree, comparing them via Clams plugin-based evaluation services. The component tree is described more in-depth in Section 4.2.2, and currently, multiple such evaluation services exist to compute and compare the possible refinement options, e.g., for various QoS like availability or cost.

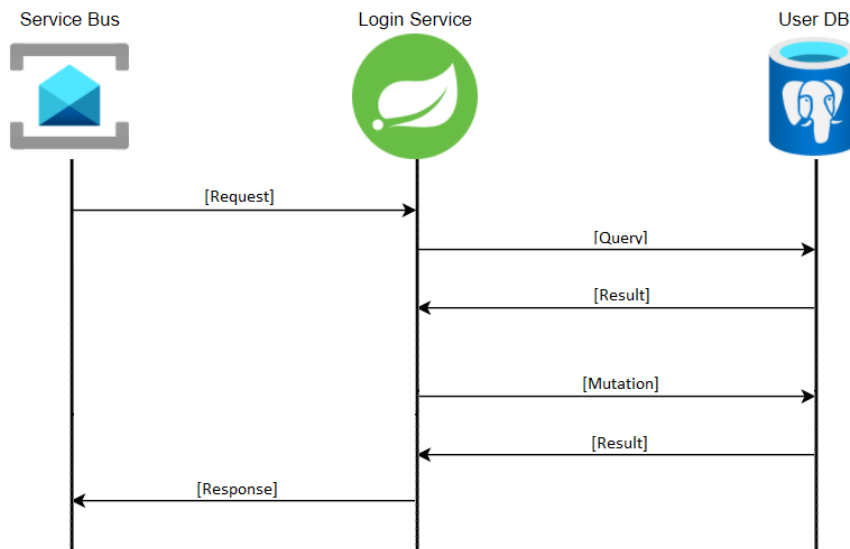
This work further extends the scope of evaluation services by additionally providing the means to analyze the predicted performance of a cloud application model under various conditions to detect design flaws that might lead to performance problems and bottlenecks that will be expensive to correct once the application is readily developed or - in the worst case - already in production.

Clams supports the modeling of multi-cloud applications by allowing the architect to use cloud components from different CSPs in the same application model. As CSPs can implement cloud patterns in different ways, each CSP has their own component tree with Clams.

### 4.2.1 Sequence Diagram

Clams SQDs follow the notation of the MSC diagram brought forward by Uchitel et al. [UKM03], [UKM04] and describe the interaction between cloud components occurring when a user interacts with the cloud application. Figure 4.1 shows a visual representation of an exemplary SQD - the example models a *Login* functionality with three interacting components. The most left component is the Service Bus, which transports the user request into the system. The *Login Service* receives the request, executes some arbitrary business logic, and sends a further request to the *User DB*, e.g., querying the user's credentials. The *User DB* replies with the Query-Result to the Login Service that subsequently updates the user's login status in the *User DB* and finally replies to the *Service Bus* that hands the result back to the user.

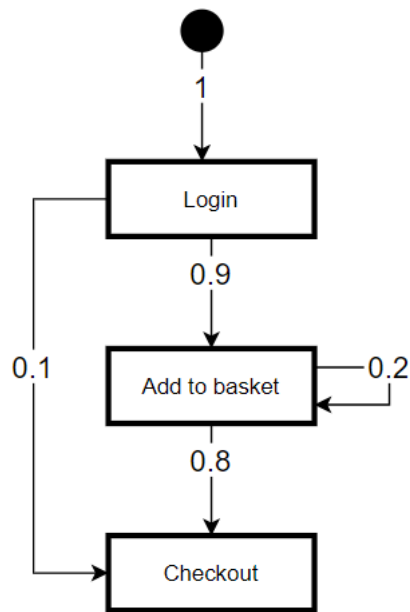
Clams does not need for the architect to provide information about messages exchanged between two components besides specifying a source component, a target component, and optionally a label. However, it is possible to add custom attributes to a message, which will become relevant later when extending the model.



**Figure 4.1:** Visual representation of an SQD in the OpenClams web interface depicting the interaction between instances of cloud components for the “Login”-State

#### 4.2.2 User Profile

The Clams UP is based on the LTS proposed by Uchitel et al. [UKM03], [UKM04] and resembles probabilistic user interaction with the cloud application. It represents a weighted graph in which the nodes are the states a user can be in at any given time, and the edges are the possible transitions between states that occur with a given probability called arrows. A UP is an absorbing Markov Chain in which a user enters the system via the starting state, called Dot, and leaves the system via end states - sinks that have no outgoing arrows. Figure 4.2 depicts an exemplary UP. In this model of a simplistic webshop, a user always starts a session by logging into the application. After logging in, there are two possible ways in which a user can continue using the application. With a probability of 90%, the user adds an item to his basket, and with a 10% chance, the user directly checks out previously added items. After a user added an item to his basket in the *Add to basket* state, there are two states for the user to transition. With a probability of 80%, the user will check out his basket, thereby leaving the system. However, with a 20% chance, a user will add another item to the basket, reentering the same state. This looping interaction is represented by an arrow that has the same source and target, here the *Add to basket* state. Naturally, the modeled behavior is simplified, lacking the possibilities for a user to register before logging in or to browse items before adding them to the basket.



**Figure 4.2:** Visual representation of a User Profile in the OpenClams web interface depicting the interaction of a user with a simplified webshop application

### State

States of a UP act as a reference to an SQD. While an SQD describes *how* to carry out the interaction between cloud components of an application that are necessary to realize a certain user interaction, a state specifies *when* the system must execute the interactions of an SQD. A state has a label, which displays the name of the linked SQD. For states that are neither the starting state nor an absorbing state, at least one incoming arrow and at least one outgoing arrow must be present. Further, the sum of probability on all outgoing arrows of a state must accumulate to exactly one, which means that a user leaves the state with a probability of 100%. In contrast, an absorbing state can only have incoming arrows but can not have outgoing ones.

A special state is the starting state which is represented by a dot in the UP. It is the only state that can not have incoming arrows and does not reference an SQD.

### Arrow

In a UP graph, the edges are called arrows and represent a connection between two states. One of the two states represents the source called  $s$ , and one the target  $t$ . An edge also has a weight  $x$  with  $0 < x \leq 1$ . The weight of an arrow poses as the transaction probability from  $s$  to  $t$  in percent, meaning that if a user is in state  $s$ , they will be in state  $t$  after the next interaction with a probability of  $x$ .



## Component Tree

In an SQD, the components can have different levels of abstraction. The most abstract form of a component represents a cloud pattern as presented in [FLR+14]. The root of a component tree is always a cloud pattern, while the children that are not leaves can be either an abstracted cloud offering, called an Abstract Component, or a further refined cloud pattern. At last, the leaves of a component tree are specific cloud offerings, called services, that can be used in a final cloud application.

An instance of such a component tree is depicted in Figure 4.3. This reduced example shows the possibilities of refinement of a Stateful Component Pattern into a final service via its children. In Clams, it is possible to traverse the component tree in any direction to refine or abstract any given component in the model. For the component tree from Figure 4.3, abstract components and services of the Azure Cloud<sup>2</sup> were used. It only serves an exemplary case, and in practice, offerings from other CSPs, like Google Cloud<sup>3</sup> or Amazon Web Services (AWS)<sup>4</sup> could be used. In theory, it is also possible to use cloud offerings from different CSPs in the same Clams model.

## Instances

Instances represent an identifiable, specific resource based on a component from the Clams component tree, and different instances can use the same component. For example, different instances of the *Azure Database for PostgreSQL/Basic/Gen5 4v Core* Service from Figure 4.3 can be used to model a *User DB*, *Order DB*, *Product DB*, and more in the same Clams model. Further, it is possible to reference the same instance in different SQDs. An example for this would be to reference the *Product DB* in an SQD modeling the *Browse items* state and in an SQD modeling the *Buy items* state as both need to access the same *Product DB*. These instances use the same component from the component tree *and* represent the same instance of this component. This means that requests to that component in both SQDs model a request to the same resource, although it does not necessarily mean that the requests use the same endpoint because one cloud component can offer multiple endpoints.

For example, in the SQD that models the *Browse items* user action, a request to the *Product DB* instance might represent a query to check if the product is currently available, while in the *Buy items* SQD, the request to the same instance could be a mutation reducing the stock of a previously reserved item.

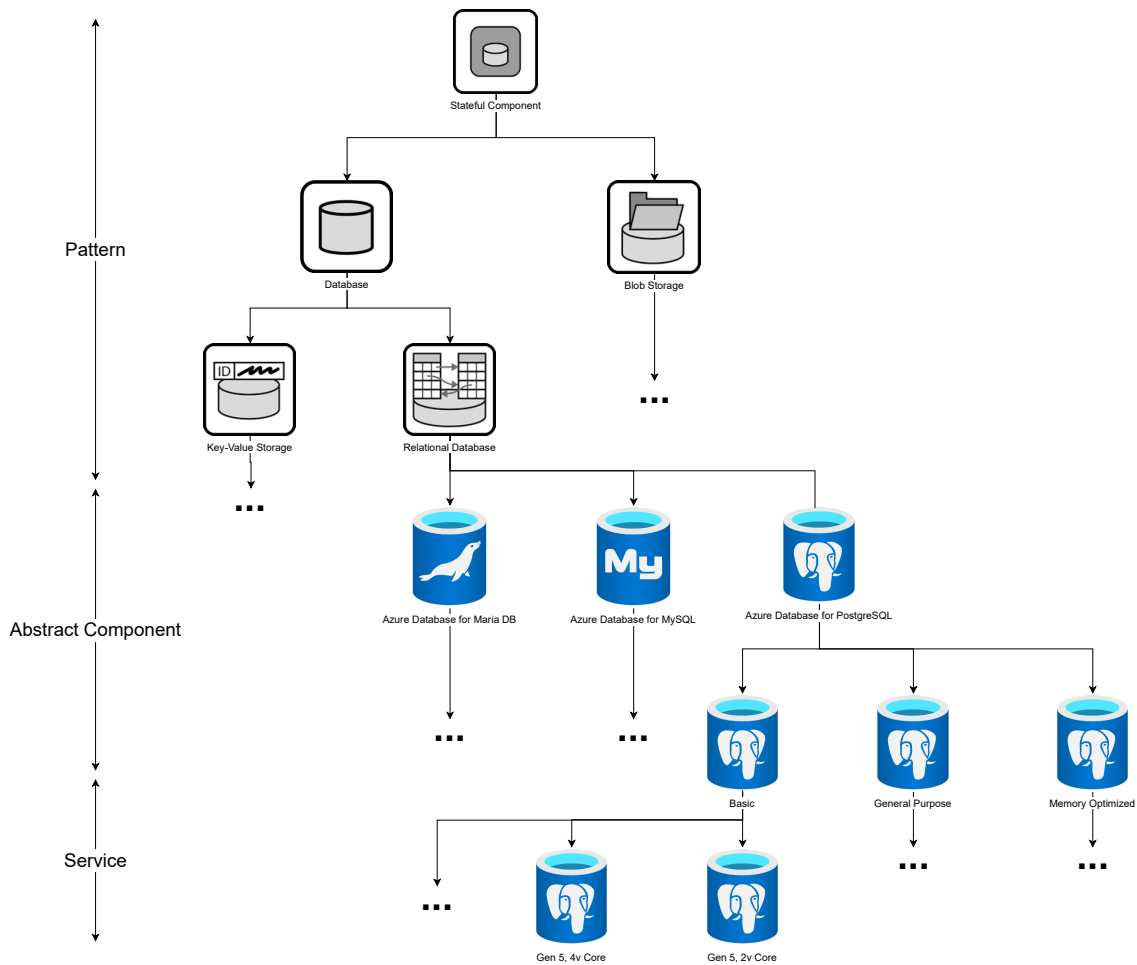
## 4.3 Layered Queuing Network

The LQN model has been proposed by Woodside et al. [RS95], [WR96], [Woo89] and describes a canonical form for layered extended queuing networks. As a result of a request from a higher level, servers at one level make requests to servers at lower levels, resulting in the layered structure. LQNs main use case is in SPE to analyze the performance of software system models.

<sup>2</sup><https://azure.microsoft.com/>

<sup>3</sup><https://cloud.google.com/>

<sup>4</sup><https://aws.amazon.com/>

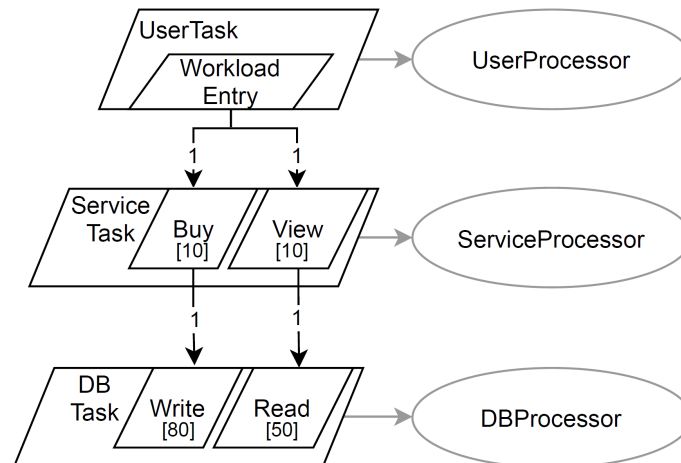


**Figure 4.3:** Clams component tree for the *Stateful Component* pattern and its refinement options

### 4.3.1 Structure

An LQN consists out of three main components; processors, tasks, and entries. These elements make up the various layers communicating with each other in different manners. When the LQN model was proposed, it did not have the cloud computing domain in mind. Due to this, it is necessary to model hardware resources like the Central processing unit (CPU) of servers that the components in the LQN are running on. This information about the components hardware is typically not available when it comes to cloud applications as software engineers no longer have control over the bare metal on which the application’s components will be deployed.

For this reason, assumptions about the hardware layer has to be made in order to represent cloud components that will be discussed in the following section. More precisely, the following section will describe the structural elements of an LQN relevant for the later parts of this work. The elements are depicted in Figure 4.4, where the processors are depicted as ovals, the tasks as parallelograms, and the entries as smaller parallelograms contained within the tasks. Finally, in the example, the simplified structure of an LQN that represents a simple webshop, where a user can buy and view items, is shown.



**Figure 4.4:** LQN with processors, tasks and entries

### Processors and Tasks

Processors represent hardware resources in the form of available CPU that is used by logical resources called tasks. They have a queue that requests will enter if they can not be processed on arrival. Tasks are the worker threads of a processor and consume service time when accepting a request. Tasks of a processor have a multiplicity that defines how many threads can run concurrently on the processor at any given time. As the cloud offers almost indefinite hardware resources, processors and tasks will be viewed as a unit, as the number of maximum concurrently running tasks is used as an upper bound of the scalability for cloud resources in this work instead of the available processor capacity. When talking about a task in an LQN model, it is therefore always implied that a processor of infinite capacity is attached, meaning that hardware processing power is always available if a task gets activated. Therefore, a request only needs to wait in the queue for its execution is if the task receiving the request is modeled only to allow the concurrent execution of a limited number of instances, all of which are already busy.

### Entries

Entries represent the services offered by the task they belong to. An entry can serve a request by directly responding to it or making subsequent requests to other entries. Serving a request takes service time on the processor, in which an entry blocks the task it belongs to. Requests between entries can be either synchronous, asynchronous, or forwarding. For this work, only synchronous and asynchronous requests are considered.

### Activities

Activities make up the internal behavior of an entry. Per default, an entry implicitly contains a single activity that takes some service time and then sends a reply to the caller. However, activities can also be used to explicitly model more complex behavior of an entry, like calling other entries.

### **Precedence**

Precedences can be used to connect two or more activities. They make it possible to model the more complex behavior of an entry by creating an activity graph that consists of various operations such as a simple sequence, joins, forks, and a loop.

### **4.3.2 Layered Queuing Network Solver**

The Layered Queuing Network Solver (LQNS) is a tool developed at the University of Carleton [FMW+05] that enables the analytical solving of LQN models described in a standardized input format. As a result, it provides various performance parameters for the LQN model and its components. The parameters that are for this work are described in the following.

### **Utilization**

The utilization of each processor, task, entry, and activity is given as the number of average active resources. For example, if only a single task is available and active for half of the total execution time, its utilization is 0.5, meaning that it is active for 50% of the time. The utilization of a resource can also be greater than one if there is more than one available resource. For example, if the multiplicity of a task is ten and, on average, only two of the tasks are active, then the resulting utilization of that task, calculated by the LQNS, will be 2.

### **Mean Delay**

In the result provided by the LQNS, the mean delay of a request made between two entries or from an activity to an entry describes how much time it had to wait in the queue of the requested resources on average. Thus, the mean delay is a good way of measuring the severeness of a bottleneck. The greater the mean delay, the worse the bottleneck.

## 5 Concept & Implementation

This chapter describes the concepts of extending the Clams modeling concepts by introducing performance annotations and subsequently transforming it into an LQN model for conducting a performance analysis using standardized tooling. The chapter also presents the implementation as a Node.js evaluation service that takes in a performance-annotated Clams model with some additional configuration and outputs the results of the performance analysis, including the utilization of a Clams models instances and potential bottlenecks and their severeness.

### 5.1 Concept

Several proposals have been made for transforming various UML models into an LQN model to enable conducting performance analysis on it [Isr01][PW05][PS02][GP05]. These works provide algorithms and solutions for the transformation of a single diagram type into an LQN model. In the case of Clams models, they consist out of two different diagram types in the form of UPs, modeling how a user interacts with the system, and SQDs, modeling which services of the system need to communicate for any given user interaction. The following part of this section describes how the different parts of a Clams model can be extended and used to generate the workload and service layer of the LQN performance model, which can later be analytically solved to predict the impact on each service of the application and to detect possible bottlenecks.

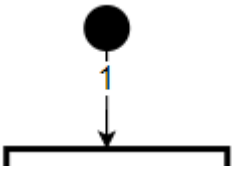
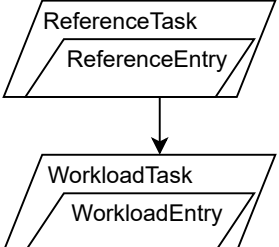
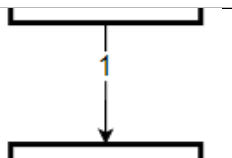


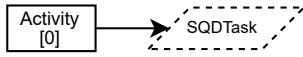
#### 5.1.1 Workload layer

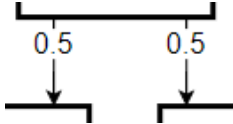
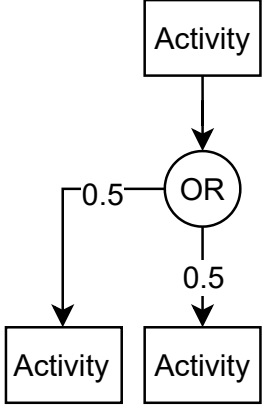
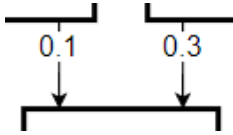
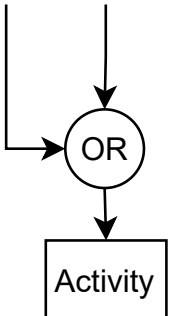
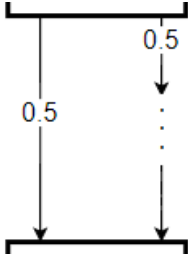
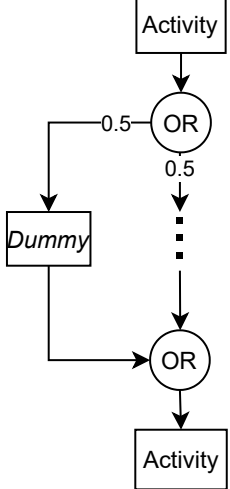
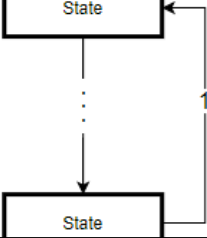
In Clams, the user interaction with the application is modeled via state transitions and their probabilities in UPs. In the LQN model, the workload is modeled via a so-called reference task, which does not accept requests and instead only makes requests to the lower layer, thereby creating a workload on the system. Usually these reference tasks consist of a single entry that simply calls the first entry of the system layer. This can not represent a stateful user behavior as modeled in a Clams UP, where a user in a certain state is only able to transition to a limited number of other states. Therefore, it has been proposed to add another layer in form of a task to model more complex user behavior. This additional task is called workload task and uses an LQN activity graph to make sequential or parallel calls to the service layer.

In Figure 5.1, the UP from Figure 4.2 has been transformed into an LQN workload layer consisting of a reference task and a workload task, as will be described in the following sections. Furthermore, the service layer in Figure 5.1 is shortened only to contain the initial tasks of each SQD, as it is described in-depth in Section 5.1.2.

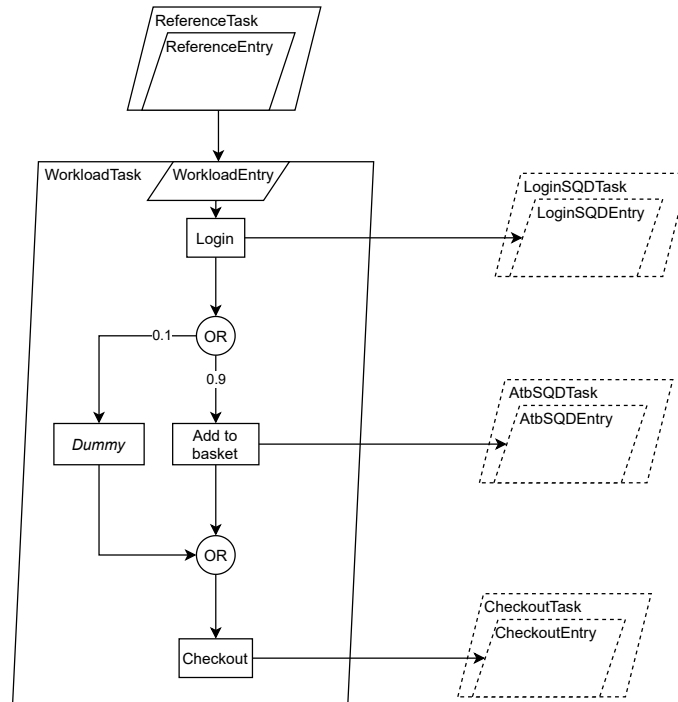
**Transforming UP to LQN elements**

The LQN components used to represent each UP component can be found in Table 5.1. As a separate task and processor are used as a container for the user interaction that creates the workload on the service layer, a workload processor, task, and entry are used as a container for the further UP flow. This UP flow always starts with a Dot, carrying no more information than marking the start of a UP. A Dot is represented in the LQN as an initial reference task along with a dummy processor and an entry that represents users entering the system at a given rate and will be described in detail in Section 5.1.1 when discussing the parts that are required for LQN models but not present in a UP diagram. The nodes of a UP are called states and will be represented in the workload entry as LQN activities. As each state in a UP represents a link to an SQD, an initial call from the activity to the first SQD entry is also part of the representation of a state in the LQN model. An arrow connecting two states in the Clams model is a simple sequential connection between the activity representing each state. If a state has multiple outgoing arrows, then the activity in the LQN needs to be followed by an OR-fork as the probabilistic semantic of Clams means that only one of the following states can be entered time with the given probability of each arrow. Thus, the arrow's probability is represented as the weight of each outgoing call of the OR-fork. If a state has multiple incoming arrows, this will be represented by an OR-join preceding the activity in the LQN model. Suppose a state with multiple outgoing arrows is connected to a state with multiple incoming arrows. In that case, a *Dummy* activity has to be added to the connection between the two, as is depicted in Figure 5.1 for the connection between the *Login* state and *Checkout* state from Figure 4.2. That is the case because, in an LQN model, it is not possible to directly connect two precedences like the OR-fork of the *Login* state with the OR-join of the *Checkout* state. The *Dummy* action has no other purpose beyond proxying the connection to enable a transition of the UP semantic into the LQN model.

UP component	Representation	LQN component(s)	Representation
Dot		Reference task, Reference entry, Workload task, Workload entry, Request	
Single Arrow		Sequence	
State		Activity without service time, Sync request to initial SQD task	

Multiple outgoing arrows		OR-fork	
Multiple incoming arrows		OR-join	
Multiple outgoing followed by multiple incoming arrows		OR-fork, Dummy activity, OR-join,	
Recursive loop		-	

**Table 5.1:** Transformation of Clams UP components to LQN components and their visual representation



**Figure 5.1:** LQN workload layer representing the UP from Figure 4.2 created by following the transition rules from Table 5.1

### Recursive loops

A structure that can not be represented in an LQN model but may be present in a Clams UP is recursive loops. Recursive loops occur because of the possibility of adding backlinks to a UP - arrows pointing to already visited states. The LQN model only supports closed loops with a fixed number of iterations via the loop precedence. Previous works have proposed to use this precedence with an additional loop-task to model looping behavior [KR08][PW05]. This approach is not feasible for the Clams UP, as it would lead creating a loop-tasks chain of infinite length due to the UPs recursive nature. To still support the analysis of Clams models that contain such loops, a different approach to handle this problem has been taken in this work. By eliminating arrows that point to already known states and redistributing the weight of the removed edge to each of the remaining outgoing arrows so that the state keeps the sum of probabilities of its outgoing arrows at 1, this allows the removal of recursive calls while keeping the resulting LQN model valid.

The approach naturally leads to somewhat different results because of the change in the UPs structure. It is thus discouraged to use backlinks in UPs if it is to be used for performance analysis. The cloud architect using the evaluation service developed in this work needs to be made aware of the possible flawed results if a UP containing recursive loops is given as an input.

The result of removing a recursive loop is depicted in Figure 5.1. Following Algorithm 5.1, the backlink from the *Add to basket* state with a probability of 0.2 has been removed. As this leaves the state with only one outgoing arrow left, it results in a weight of 1 for the arrow from the *Add to basket* state to the *Checkout* state.



---

**Algorithm 5.1** Removing backlinks and redistributing branch weights starting from the dot state

---

**Input:** Starting state “dot”

```

1: removeBacklinks(dot, [dot])
2:
3: procedure REMOVEBACKLINKS(state, previousStates)
4:   for outgoingEdge of state.outgoingEdges do
5:     if outgoingEdge.target in previousStates then
6:       remove(outgoingEdge)
7:     else
8:       previousStates.add(state)
9:       removeBacklinks(outgoingEdge.target, newStates)
10:    end if
11:  end for
12:  redistributeWeight(state)
13: end procedure
14:
15: procedure REDISTRIBUTEWEIGHT(state)
16:   edgeWeightSum  $\leftarrow$  sum(state.outgoingEdges.weight)
17:   for outgoingEdge of state.outgoingEdges do
18:     outgoingEdge.weight  $\leftarrow$  outgoingEdge.weight  $\div$  edgeWeightSum
19:   end for
20: end procedure

```

---

### Workload attributes

The previous sections described how the states and arrows of a UP can or can not be transformed into the workload layer of an LQN model, more precisely, a reference task and a workload task. The LQN model contains further details regarding the workload layer beyond the pure structure of its activities. These details, listed in Table 5.2, can be provided and tweaked by the cloud architect as additional input on a per-evaluation basis when using the evaluation service. For example, the arrival rate in milliseconds specifies the average time between two users entering the system. This arrival rate can either describe an open or closed workload. An open workload means that a user arrives after the amount of time specified in the arrival rate and a closed workload means that there is a specified number of users that reenter the system after the time specified in the arrival rate. Further, an average user think time can be taken into account. Usually, in an LQN model, think time can be set as a parameter to the reference task, like the arrival rate, but since the reference task is only used to signal users entering the system and the interaction of the user with the service layer is entirely contained in the workload layer, this is not possible. Instead, the think time will be modeled by adding additional *think time* activities after each workload activity representing a UP state in the workload task with its service time matching the provided average think time. The average network delay can also be taken into account. This number describes the milliseconds that it takes a user interaction to reach the system.

Workload attribute name	Value type	LQN representation	Required	Default
Arrival rate	milliseconds	Reference task attribute	Yes	-
Workload type	OPEN/ CLOSED	Reference task attribute	No	OPEN
User amount	integer	Reference task multiplicity	For CLOSED workload type	-
Avg. think time	milliseconds	Delay activity following state activity	No	0
Avg. network delay	milliseconds	Service time of SQD start task	No	1

**Table 5.2:** Workload attributes to be provided additionally to the Clams model

### 5.1.2 Service layer

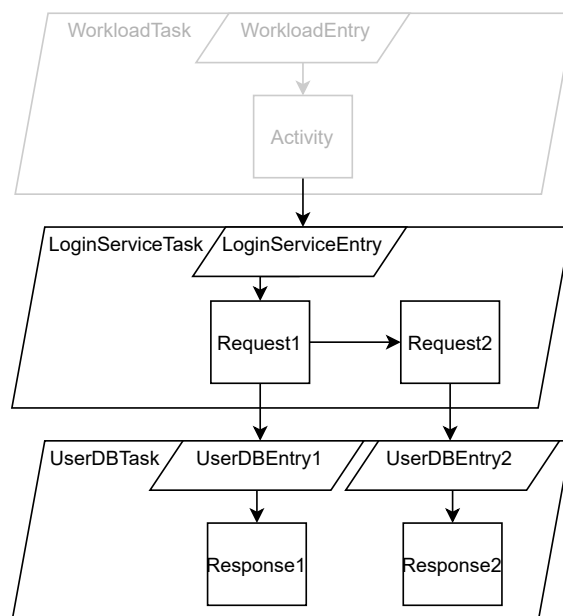
The interaction between instances of components in Clams is modeled via SQDs. An SQD describes these interactions via messages exchanged between instances in a point-to-point manner. This message exchange can either be of synchronous or asynchronous nature. A synchronous message signals that the requesting instance is blocked until a response to the sent request is received. In contrast, an asynchronous message is a simple invocation of another instance without the requesting instance awaiting any produced result, meaning that it is freed right after the asynchronous request is sent. All of these SQD parts can be represented in an LQN model, as depicted in Table 5.3. The LQN notation allows for an entry to stand alone - without containing any activities - if the entry is simple. A simple entry is an entry that takes service time and responds right after. Only for entries that employ more complex behavior, activities must be used to model the entry's steps. Because, in an SQD, an instance can make any number of requests when invoked, which a simple entry can not represent, the activity notation is always chosen over the simple stand-alone entry notation. In Figure 5.2, this can be seen in the *LoginServiceTask*, where the two subsequent requests to the *UserDBTask* are modeled as two sequential *Request1* and *Request2* activities. Also, in the *UserDBTask*, where a stand-alone entry could have been used to model the simple direct responding behavior, an activity is used to send the response.

### Performance attributes

Performance and service times have not been a part of Clams SQD models so far but are necessary for the service layer of LQN models. In order to introduce the performance aspect and to enable the performance analysis of a Clams model, this work adds some extensions in the form of annotations to Clams modeling capabilities when it comes to SQDs. SQD instances and messages are extensible via so-called meta attributes that can hold any key-value information. The additional information in the form of meta attributes that SQD components need to be annotated with to enable a transformation into the LQN service layer and thus performance analysis is listed in Table 5.4. For a message, the average service time it imposes on the called instance needs to be provided. This provided integer

SQD component	Representation	LQN component	Representation
Instance		Service task, Service entry	
Async request		Async request from activity to entry	
Sync request		Sync request from activity to entry	
Sync response		End of call chain	

**Table 5.3:** Transformation of SQD components to LQN components



**Figure 5.2:** LQN service layer representing the SQD from Figure 4.1 created by following the transition rules from Table 5.3

Meta attribute name	Value type	Clams component	required	default
Avg. service time	milliseconds	Message	Not for replies	-
Message type	SYNC/ASYNC	Message	Yes	-
Message type sync	REQUEST/REPLY	Message	If msgType is SYNC	-
Max. concurrency	integer	Instance	No	$\infty$

**Table 5.4:** Performance annotations of instances and messages in a Clams SQD

will be the service time of the entry that represents the called instance in the resulting LQN. Further, it needs to be specified if a message is part of an asynchronous or synchronous interaction and, if the exchange is synchronous, both messages need to be labeled as the request or the reply of said synchronous exchange.

For instances in an SQD, the only information that needs to be provided as an annotation in its meta attributes is the maximum concurrency. This information is optional, as the default concurrency for instances is infinite concurrency due to the previously stated theoretical unlimited horizontal scalability of stateless components. This means that the attribute only needs to be set for instances that represent stateful components.

### Transforming multiple SQDs

Unlike the UP, there typically is more than one SQD contained in a Clams model. More precisely, as each state in a UP references an SQD, there is an SQD contained for each state. All of the different sub-models of the Clams model need to be transformed into a single LQN model. In Section 5.1.1, Figure 5.1, and Figure 5.2 it has been shown how the UP, represented as the workload layer in the resulting LQN, is connected to its referenced SQDs, represented as the service layer in the resulting LQN. In the following it will subsequently be discussed how multiple SQDs can be transformed into a single service layer.

It has been stated that multiple SQDs can share any amount of identical instances. Two SQDs containing the same instance describes that the same resource is part of two different interactions in a Clams model. For example, the *Registration* SQD from Figure 5.3 and the *Login* SQD from Figure 4.1 contain the same *User DB* instance. This is the case because when a user logs into the system, the *Login Service* makes a request to the *User DB* to validate the provided credentials. The same database needs to be mutated if a user registers, as the DB needs to be mutated to hold the information for the newly created user account. It can be seen that even though the instance making the request is different in each SQD, the data source in the form of the *User DB* is the same resource. A resource is represented as a task in an LQN model, which means that each instance - pointing to the same resource - is also modeled as a single task in the resulting LQN model following the transformation algorithm described in Algorithm 5.2. Like the multiple requests made to the same instance in Figure 4.2 that are represented as multiple entries in the same task in Figure 5.2, the requests to the same instance from different SQDs are represented as multiple entries in the same task.

Figure 5.4 depicts the task in the resulting LQN model representing the *User DB* instance, assuming both the *Login* and *Registration* SQD to be contained in the same UP. In this example, it is assumed that the requests from the *Login* SQD take 30 and 50 milliseconds of service time on the *User DB*,

**Algorithm 5.2** Creating and connecting tasks, entries for multiple SQDs

---

**Input:** List of SQDs “sqds”

```

1: instanceTasks ← []
2: for sqd of sqds do
3:   startTask ← createTaskAndEntryAsSQDStart(sqd)
4:   connectToWorkloadActivity(startTask)
5:   entries ← []
6:   entries.add(startTask.getEntry())
7:   for message of sqd.messages do // messages are ordered sequentially
8:     sourceTask ← instanceTasks.getOrCreateForInstance(message.source)
9:     targetTask ← instanceTasks.getOrCreateForInstance(message.target)
10:    sourceEntry ← entries.getNotBlockedOrCreateForInstance(message.source)
11:    targetEntry ← entries.createForInstance(message.target)
12:    if message.msgType == SYNC then
13:      if message.msgTypeSync == REQUEST then
14:        sourceEntry.appendActivitySyncCallTo(targetEntry)
15:        sourceEntry.blocked ← true
16:      else if message.msgTypeSync == RESPONSE then
17:        targetEntry.blocked ← false // entry can make further requests
18:      end if
19:    else if message.msgType == ASYNC then
20:      sourceEntry.appendAyncActivityCallTo(targetEntry)
21:    end if
22:    sourceTask.save(sourceEntry)
23:    targetTask.save(targetEntry)
24:  end for
25: end for

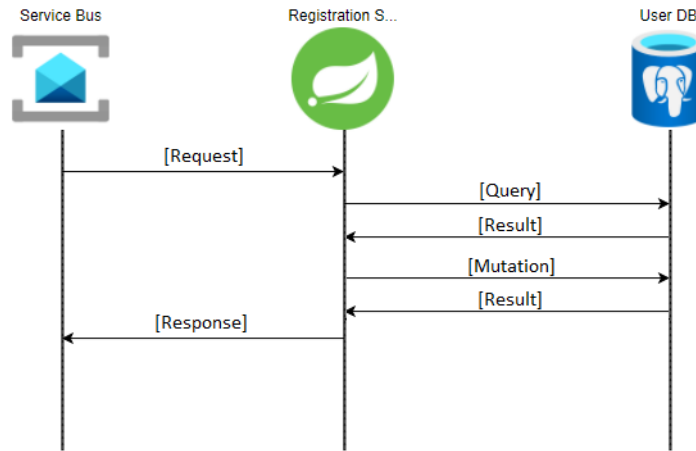
```

---

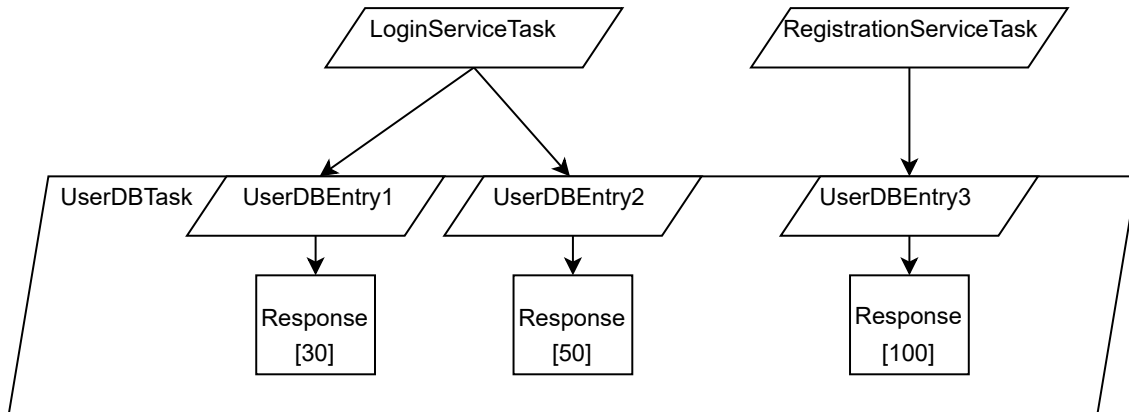
and the request from the *Registration* SQD takes 100 milliseconds. It can be seen that all entries representing the different requests are contained within the same task, and therefore the service time is imposed on the same resource.

## 5.2 Implementation

The concepts proposed in Section 5.1 have been implemented as an evaluation service extending the Clams ecosystem. More precisely, the OpenClams web application has been extended, and a Node.js service in the Typescript programming language has been developed. The evaluation service takes a Clams model in the Javascript Object Notation (JSON) format, containing the performance annotations from Table 5.4 and the additional workload parameters from Table 5.2 as an input. The evaluation service then transforms the input into a Typescript object representing the LQN model to finally generate the input file for the standardized tooling that is the LQNS in order to solve the performance model. The following sections will describe these parts of the implementation.



**Figure 5.3:** Registration SQD using the same *User DB* instance as the *Login SQD* from Figure 4.1



**Figure 5.4:** Resulting LQN service layer when transforming a Clams model using the same *User DB* instance in multiple SQDs

### 5.2.1 OpenClams

OpenClams is an open-source reference implementation for the Clams model and provides tooling for creating and manipulating these models. One of these provided tools is a Graphical User Interface (GUI) in the form of a web application, allowing the cloud architect to manage their Clams models in a user-friendly way. In this work, the GUI has been extended in multiple ways. New meta attributes have been introduced for the message and instance components of the SQD graph that can be added and removed in the GUI as is depicted in Figure 5.5 for the *User DB* instance and in Figure 5.6 for a message. Further, a new entry in the dropdown menu for selecting an evaluation service has been added, as can be seen in Figure 5.7. Selecting this option from the menu then opens the dialog depicted in Figure 5.8 which allows the cloud architect to provide the workload attributes from Table 5.2 and subsequently start the performance evaluation.

**Azure Database for PostgreSQL**

Name  
User DB

Change the name of the instance.

**Meta Data**

+ Add Data

Key	Value
maxConcurrency	50

**Figure 5.5:** The avgConcurrency meta attribute added to the *User DB* instance

**TCP Connection**

**Meta Data**

+ Add Data

Key	Value
avgTimeMs	30
msgType	SYNC
msgTypeSync	REQUEST

**Figure 5.6:** Message meta attributes of a synchronous request between two instances

**Evaluate**

- Predict Availability
- Predict Performance

**Figure 5.7:** Additional “Predict Performance” option in the dropdown menu for selecting evaluation services

Analyze predicted performance for workload

Provide workload arguments

Arrival rate in ms

Avg. user think time in ms (optional)

Avg. network delay in ms (optional)

Workload type [OPEN/CLOSED](optional)

User amount (optional)

Cancel Create

**Figure 5.8:** Dialog asking for the workload attributes for an invocation of the performance evaluation service

Starting the performance evaluation sends a Hypertext Transfer Protocol (HTTP) POST-request to the address that the evaluation service is running at. The requests body contains the Clams model and the workload attributes. The Clams model provided to an evaluation server by the cloud architect is expected to be in the OpenClams JSON format. An abbreviated example of a clams model persisted in this JSON format is displayed in Listing 5.1. The example JSON depicts the serialized graph from Figure 4.1 and Figure 4.2, with only the first entries of each JSON array being displayed.

OpenClams provides tooling to deserialize data provided in the presented JSON format into a Typescript object. The tooling is in the form of a Node Package Manager (npm) module called ClamsML<sup>1</sup> and is part of the OpenClams environment. The ClamsML module provides the functionality to query and mutate a deserialized Clams model.

### 5.2.2 Clams object to LQN object transformation

Because the ClamsML npm package is available only for the Node.js/Typescript environment, the program developed as part of this thesis was also developed using the Typescript programming language on the Javascript/Typescript runtime Node.js.

In order to represent the LQN model as an object in Typescript, classes for every structural element of an LQN have been created. Additionally, for each step in the transformation, a class that instantiates and connects objects of the structural classes by providing the necessary information from the Clams model or the workload attributes has been developed.

After deserializing the JSON that has been provided as an input into the Typescript object representation of the Clams model, it is possible to access all parts of the model via the ClamsML package. In the first step, all recursive loops are removed from the UP following the algorithm from

---

<sup>1</sup><https://github.com/openclams/clams-ml>



Algorithm 5.1. Subsequently, an empty LQN object is created. A reference task and a workload task containing the given workload attributes are added to the LQN object in the next step. Then, the program iterates through all SQDs in the Clams model, as described in Algorithm 5.2, and builds the service layer with all its tasks, entries, and activities using the “avgTimeMs” performance annotation as an activities service time. At last, the start task of each SQD is connected to its associated activity in the workload task to finalize the LQN object.

## 5.3 Solving the Layered Queuing Network model

As a design decision, it was decided to model the LQN object without considering the solver that will later be used to solve the LQN model contained in its object representation. This has the advantage that in order to use another LQN solver only the program used to transform the LQN object representation into the input format used for the specific solver needs to be adjusted or newly developed. Most importantly, this means that the transformation from the Clams model into the LQN model does not need to be changed when using another LQN solver as it would be the case if a direct transformation of the Clams model into the LQN solvers input format would be carried out instead. In summary, it can be said that the architecture is based on the loose coupling paradigm leading to no correlation between the Clams model and the input format of the used LQN solver. For this thesis, the LQN solver proposed and developed by Franks et al. [FMW+05], called LQNS, was used to solve the LQN model. The following sections describe how the transformation of the LQN object representation into the LQNS input format is conducted and finally how the results provided by the LQNS are parsed and returned to the OpenClams user.

### 5.3.1 Transformation of the LQN object into the LQNS input format

The input format for the LQNS is described in its user manual [FMW+05]. It is a plain text format describing an LQN via different abbreviations in a non-human-readable syntax. The user manual also contains a syntax definition in the form of transitional rules in Backus-Naur form (BNF) that are typically used to describe formal languages. The resulting document consists of different sections on the top layer, one for each type of LQN element. In the following the generation and structure of each section will be described.

#### Processors & Tasks

As in this work processors and tasks of an LQN model are only considered as a unit all the information about both is contained in the tasks of the LQN object. For creating processors and tasks in the LQNS input file, the algorithm loops through all tasks and transforms them to the reference task, the workload task, the tasks with infinite multiplicity, and the tasks with limited multiplicity. Additionally, to each task definition, a processor definition is added.

In Listing 5.2, the representation of processors and tasks in the LQNS input format is shown. In line 1, the beginning of the processor definition segment is signaled by a  $P$ , followed by the total number of processors that are defined in the following lines. In line 2, it can be seen that the  $p$  signals a

processor definition. The  $i$  signals that the processor has infinite multiplicity.

In line 6, the task definition segment is started with a  $T$ , followed by the total number of tasks. In lines 7 to 10, a task for an open arrival reference task, closed reference task, finite multiplicity task, and infinite multiplicity task is defined. Each line starts with a  $t$  that signals the definition of a task followed by the task name, which is “reference\_task” for the reference task, “workload\_task” for the workload task, and the instance name followed by “\_task” for any task representing an SQD instance. Following the task name there are two qualifiers  $r$  and  $n$  that signal if the defined task is a reference or a non-reference task. The  $\$entryList$  variable represents a whitespace-separated list of entry names contained in the task. The following  $-l$  signals the end of a list, similar to the end of the processor and task definition segments. The  $\$processorName$  holds the name of the processor for the task - the task’s instance name followed by  $\_processor$ .

Line 7 and 8 both define a reference task. The first one is an open arrival reference task as signaled by the multiplicity of 1 and a defined  $\$arrivalRate$ . An open arrival reference task represents a single user entering the system every  $\$arrivalRate$  milliseconds. In contrast, line 8 defines the reference task for a closed workload, as the multiplicity  $m$  is set to the number of users and  $z$  represents the  $\$avgReentryTime$  rather than the arrival rate. A closed workload means that the reference task makes synchronous requests to the workload task and is blocked for the duration of all workload activities. Then after leaving the system, the user enters the system again after  $\$avgReentryTime$  in a Poisson distribution. The workload task is represented as a non-reference task with infinite multiplicity.

## Entries

For creating entries, the algorithm first creates one entry for the reference task and the workload task. Secondly, a call from the reference entry to the workload entry is created. Whether this call is of asynchronous or synchronous nature depends if the LQN models workload is set to be open arrival or closed. The entry of the reference task is the only entry that directly calls another entry, as all entries of other tasks contain activities that further specify their behavior. This can be seen in Listing 5.3, which represents the definition of the entries in the LQNS input format. Line 1 and 5 contain the typical segment wrapper led by an  $E$ . Line 2 to 4 depict the definition of the reference tasks entry. The line starting with an  $s$  specifies the entry with a service time, set to the smallest possible number accepted by the LQNS, as the reference entry should not have a service time but the LQNS tool does not allow a service time of 0 for entries that have no explicit activities. The 3rd and fourth lines show the definition of a synchronous and asynchronous call from the entry that’s name is contained in the  $\$entryName$  variable to the entry with the name  $\$targetEntryName$  and a weight representing the number of calls, that is always 1 for this work.

Finally, line 5 represents the entry definition for the workload task and all instance tasks, as the  $A$  specifies an entry to contain activities that further define its behavior. Such an entry contains an  $\$entryName$ , by which it can be referenced from other entries or activities, and the name of the root activity of its activity graph.

## Activities

The final part of the LQNS input format is the activity definition segment in which the activities, their service time, and their interaction between each other and other entries is defined. The syntax for defining activities is different from other parts of the LQNS input format in that the top line of the segments wrapper does not contain the number of LQN components defined in the segment but rather an *A* followed by the *\$taskName* of the task the subsequently defined activities belong to as can be seen in Listing 5.4. Listing 5.4 further shows how the activity definition segment is separated by a *colon*. The part before the separator describes the activities with their service times and calls to entries. The notation is similar to the entry notation from Listing 5.3, with an *s* defining the service time, *y* a synchronous call, and *z* an asynchronous call. In contrast to the entry definition, it is possible not to specify a service time for activities that make requests to entries.

The second part after the *colon*-separator describes the activity graph structure. In line 6, the notation of a simple sequence between two activities is shown. The OR-fork and OR-join operators can be seen in lines 7 and 8. It is crucial for every line but the last one following the *colon*-separator to end with a semicolon or else the input format will not be valid.

### 5.3.2 Result parsing

After invoking the LQNS tool with the generated input file, a result file is produced that will be read and parsed by the evaluation service to extract the results and provide them to the Clams user. As described in Section 5.1, the interest of this work is in detecting bottlenecks and their severeness. This is why regarding the LQNS results, the main focus is on the “Mean delay for a rendezvous request” and the “Task Utilization” of each service task belonging to a Clams instance.

For the “Mean delay for a rendezvous request” segment, besides the mean delay, the targeted entry is of interest, as this entry belongs to a task representing a bottleneck in the LQN model and thus an instance in the Clams model. The “Utilization” of other instances is also interesting for a cloud architect to know how the workload puts strain on each instance in the system.

The resulting file consists out of many segments describing the analyzed LQN model and its parameters. In Listing 5.5, the segment representing the mean delay can be seen. Its start is represented by a *W* followed by the *\$numberOfRendezvous*. The result is parsed from there extracting the *\$taskName*, *\$entryOrActivityName*, *\$targetEntryName* and *\$meanDelay*. Further, the mean utilization per task is calculated as a sum of the utilization of all of the tasks entries that is reported as depicted in Listing 5.6.

The results are then aggregated, sorted, first by mean delay and second by utilization, for each instance and reported back to the OpenClams user.

**Listing 5.1** Annotated Clams model serialized as JSON (*contd.*)

---

```
1 {
2   "model": {
3     "graphs": [
4       {
5         "id": "Graph_0",
6         "name": "Shopping",
7         "type": "UserProfile",
8         "nodes": [
9           {
10            "type": "Dot",
11            "id": "Dot0",
12          },
13          {
14            "id": "State1",
15            "type": "State",
16            "sequenceDiagramId": "Graph_1",
17          }
18        ],
19        "edges": [
20          {
21            "type": "Arrow",
22            "from": "Dot0",
23            "to": "State1",
24            "p": 1,
25          },
26        ]
27      },
28      {
29        "id": "Graph_1",
30        "lastId": 5,
31        "name": "Login",
32        "type": "SequenceDiagram",
33        "nodes": [
34          {
35            "type": "Instance",
36            "id": "Instance0",
37            "component": "Service Bus"
38          },
39          {
40            "type": "Instance",
41            "id": "Instance2",
42            "component": "Login Service"
43          }
44        ],
```

---

**Listing 5.1** Annotated Clams model serialized as JSON

```

1     "edges": [
2     {
3         "type": "Message",
4         "from": "Instance0",
5         "to": "Instance2",
6         "position": 6,
7         "edgeType": {
8             "name": "TCP Connection",
9             "attributes": [
10            {
11                "name": "avgTimeMs",
12                "value": 100,
13            },
14            {
15                "name": "msgType",
16                "value": "SYNC",
17            },
18            {
19                "name": "msgTypeSync",
20                "value": "REQUEST",
21            }
22        ]
23    }
24    },
25    ]
26 }
27 }
28 }

```

**Listing 5.2** LQN processors and tasks as represented in the LQNS input format

```

1 P $numberOfProcessors
2   p $processorName i
3   ...
4 -1
5
6 T $numberOfTasks
7   t $taskName r $entryList -1 $processorName m 1 z $arrivalRate
8   t $taskName r $entryList -1 $processorName m $noOfUsers z $avgReentryTime
9   t $taskName n $entryList -1 $processorName m $multiplicity
10  t $taskName n $entryList -1 $processorName i
11  ...
12 -1

```

## 5 Concept & Implementation

---

---

### Listing 5.3 LQN entries as represented in the LQNS input format

---

```
1 E $numberOfEntries
2   s $entryName $serviceTime -1
3   y $entryName $targetEntryName $weight -1
4   z $entryName $targetEntryName $weight -1
5   A $entryName $initialActivityName
6 -1
```

---

---

### Listing 5.4 LQN activities as represented in the LQNS input format

---

```
1 A $taskName
2   s $activityName $serviceTime
3   y $activityName $entryName $weight
4   z $activityName $entryName $weight
5 :
6   $activityName -> $targetActivityName;
7   $activityName -> ($prob1)$targetActivityName1 + .. + ($probN)$targetActivityNameN;
8   $($prob1)$activityName1 + .. + ($probN)$activityNameN -> $targetActivityName;
9   $activityName[$entryName]
10 -1
```

---

---

### Listing 5.5 Mean delay per rendezvous as noted in the LQNS result format

---

```
1 W $numberOfRendezvous
2   $taskName: $entryOrActivityName          $targetEntryName          $meanDelay
3   -1
4   ...
5   -1
6   ...
7   -1
```

---

---

### Listing 5.6 Utilization per task and entry as represented in the LQNS result format

---

```
1 P $processorName 1
2   $taskName $numberOfEntries 0 $multiplicity
3   $entryName          $entryUtilization          0          -1
4   -1
5   -1
```

---

## 6 Case Study

In this chapter a case study is carried out to validate the concept and implementation proposed in this thesis. First, a performance-annotated Clams model of a webshop containing a component representing an obvious bottleneck is presented. Second, the Clams model is transformed into an LQN model via the OpenClams evaluation service developed as a part of this work. In the next step, the input for the LQNS is generated. Using the LQNS tool, the bottleneck is detected in its result. Subsequently the Clams models and workload parameters are changed to validate if the evaluation service provides the expected results.

### 6.1 Model

For conducting the case study the Clams model of the webshop cloud application from Section 4.2 has been extended as is explained in the following sections.

#### 6.1.1 Clams model

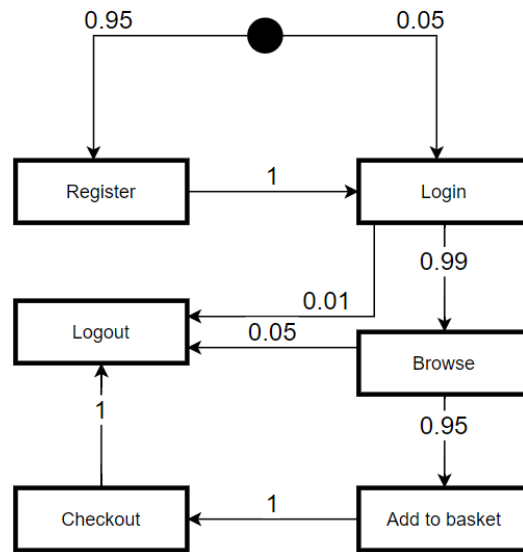
The Clams model used in this case study consists of a single UP that models the user behavior regarding a webshop application. The UP is visualized in Figure 6.1 and shows the states a user can be in after either registering or logging in. Each state is a reference to an SQD defining the communication between system components that is triggered by a user entering the state.

It is assumed that in each SQD, a *Service Bus* transports the user request into the system. The receiving service instance takes 20 milliseconds to process the request until it can make requests. The further instances and their interactions in each SQD are described in the following:

**Register** The *User Service* instance sends two following synchronous requests to the *User DB* to check for credential availability and add the user's credentials. The first request's average service time - representing a read operation - on the *User DB* is assumed to be 50 milliseconds, while the second request - representing a write operation - is expected to take 80 milliseconds of service time.

**Login** *User Service* again sends two synchronous requests to the *User DB* instance to verify the user credentials and set the user state to be logged in. The query is estimated to have an average service time of 50 milliseconds while the update's average service time is set to 60 milliseconds.

**Browse** The *Article Service* instance queries multiple articles from the *Article DB* to return to the user. The batch request has an average service time of 200ms.



**Figure 6.1:** UP modeling the user interaction with in a webshop application

**Add to basket** After processing the user request, the *Shopping Service* queries the article’s stock amount from the *Article DB* and writes to the *Order DB*, which contains the user baskets. Querying a single field is estimated to have a service time of 20 milliseconds while the write operation, again, has a service time of 80 milliseconds.

**Checkout** First, the *Shopping Service* updates the order status in the *Order DB* then it updates the article stock in the *Article DB*. Both updates have an estimated average service time of 60 milliseconds.

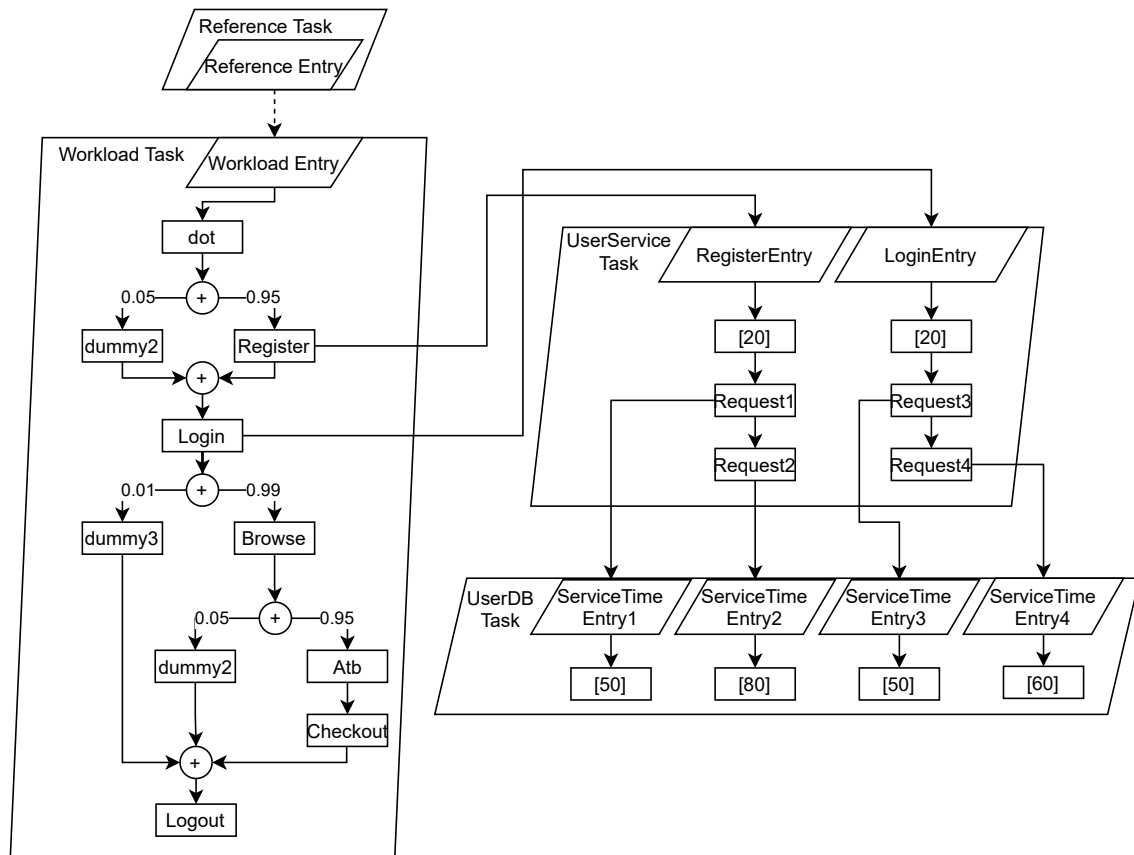
**Logout** The *User Service* changes the login state in the *User DB*, which has an average service time of 60 milliseconds

All *Service* instances represent stateless components, while the *DB* instances represent stateful components. The “maxConcurrency” performance annotation for each stateless component is set to 100 except for the *User DB*. The *User DBs* “maxConcurrency” annotation is set to 10 to represent an obvious bottleneck that is to be detected when analyzing the LQN model later.

### 6.1.2 LQN model

In the first step, the input Clams model is transformed into an LQN object. The object has been analyzed using a Javascript debugger. An abbreviated version of the analyzed model is depicted in Figure 6.2. The generated workload layer, consisting out of the reference task and the workload task, can be seen. Further, the activity graph of the workload task representing the UP from the Clams model is shown. For the service layer, only the task of the *Register* and *Login* workload activities - the *User Service* task and the *User DB* - are displayed. The calls made from the *Browse*, *Atb*, *Checkout*, and *Login* workload activities are omitted for clarity. The activities in the service layer that are labeled with a number represent a service time activity, while the *Request* activities make calls to the lower layer.





**Figure 6.2:** LQN model transformed from Clams model with shortened depiction of the service layer containing only the service tasks for the *User Service* and *User DB* instances

## 6.2 Execution

The model that has been presented in the previous section will be transformed into the LQNS input format and subsequently analyzed by the LQNS tool. First, a very low workload is chosen that is expected not to create delay for any instance in the system. Second, the workload is incrementally increased until the *User DB* instance poses as a bottleneck. Third, the “maxConcurrency” performance annotation of the *User DB* instance is incremented, and the updated model is analyzed again for the same workload attributes.

### 6.2.1 Workload attributes

The workload attributes are specified independently from the Clams model and will be defined in the following. An open workload type is chosen for the model as it is better suited for the webshop use case that serves unknown users. An example of a good use case for a closed workload would be a model of an Internet of things (IoT) platform with a given number of devices that reenter the system after a given time. The user think time and the network delay attributes are not used for this model. The arrival rate will be varied in every iteration to analyze how the model behaves under different workloads.

---

**Listing 6.1** Result of performance analysis with mean delay of 1000 milliseconds

---

```
1 [  
2 { utilization: 0.0198, componentName: 'Article Service', maxDelay: 0 },  
3 { utilization: 0.03762, componentName: 'Shopping Service', maxDelay: 0},  
4 { utilization: 0.059, componentName: 'User Service', maxDelay: 0 },  
5 { utilization: 0.13167, componentName: 'Order DB', maxDelay: 0 },  
6 { utilization: 0.27324, componentName: 'Article DB', maxDelay: 0 },  
7 { utilization: 0.2935, componentName: 'User DB', maxDelay: 0 }  
8 ]
```

---

### 6.2.2 Result analysis

The result of the program developed in this work is a list that consists of an entry for every task belonging to an instance in the provided Clams model. Each entry contains of the instances name, its maximum delay and its mean utilization.

#### Low workload

For this analysis, an arrival rate of 1000 milliseconds is chosen. This means that, on average, a user enters the system every second. Looking at the Clams model from Section 6.1.1 the sum of all service times of every request is below that. This should guarantee that the previous user has already left the system before the next user enters it, thus creating no delay for any of the instances. The analysis result is shown in Listing 6.1 and confirms that no delay occurs at any instance. Further, the utilization for each instance is less than 1, meaning that they were idle most of the time, as was expected.

#### Increasing workload

This analysis has been conducted with an increasing workload. The arrival rate has been decreased to be 500, 250, 100, 50, and 30 milliseconds.

As can be seen in Listing 6.2, Listing 6.3, and Listing 6.4, despite a linear rise in every instances utilization, no delay occurs. Only for an arrival rate of 50 milliseconds a slight delay occurs for the *User DB* component. Although, as shown in Listing 6.5, the delay is under a millisecond, this signals that the component starts to reach its maximum capacity. Finally, for the analysis with an arrival rate of 35, a delay of up to 281 milliseconds is calculated for the requests to the *User DB* component. As the average utilization of the *User DB* component approaches its maximum concurrency of 10, this makes sense as each further request can not be immediately processed and thus heavily adds to the delay.

---

**Listing 6.2** Result of performance analysis with average arrival rate of 500 milliseconds

---

```
1 [  
2 { utilization: 0.0396, componentName: 'Article Service', maxDelay: 0 },  
3 { utilization: 0.07524, componentName: 'Shopping Service', maxDelay: 0 },  
4 { utilization: 0.118, componentName: 'User Service', maxDelay: 0 },  
5 { utilization: 0.26334, componentName: 'Order DB', maxDelay: 0 },  
6 { utilization: 0.54648, componentName: 'Article DB', maxDelay: 0 },  
7 { utilization: 0.587, componentName: 'User DB', maxDelay: 0 }  
8 ]
```

---

---

---

**Listing 6.3** Result of performance analysis with average arrival rate of 250 milliseconds

---

```
1 [  
2 { utilization: 0.0792, componentName: 'Article Service', maxDelay: 0 },  
3 { utilization: 0.15048, componentName: 'Shopping Service', maxDelay: 0 },  
4 { utilization: 0.236, componentName: 'User Service', maxDelay: 0 },  
5 { utilization: 0.52668, componentName: 'Order DB', maxDelay: 0 },  
6 { utilization: 1.09296, componentName: 'Article DB', maxDelay: 0 },  
7 { utilization: 1.174, componentName: 'User DB', maxDelay: 0 }  
8 ]
```

---

---

---

**Listing 6.4** Result of performance analysis with average arrival rate of 100 milliseconds

---

```
1 [  
2 { utilization: 0.198, componentName: 'Article Service', maxDelay: 0 },  
3 { utilization: 0.3762, componentName: 'Shopping Service', maxDelay: 0 },  
4 { utilization: 0.59, componentName: 'User Service', maxDelay: 0 },  
5 { utilization: 1.3167, componentName: 'Order DB', maxDelay: 0 },  
6 { utilization: 2.7324, componentName: 'Article DB', maxDelay: 0 },  
7 { utilization: 2.935, componentName: 'User DB', maxDelay: 0 }  
8 ]
```

---

---

---

**Listing 6.5** Result of performance analysis with average arrival rate of 50 milliseconds

---

```
1 [  
2 { utilization: 0.396, componentName: 'Article Service', maxDelay: 0 },  
3 { utilization: 0.7524, componentName: 'Shopping Service', maxDelay: 0 },  
4 { utilization: 1.18, componentName: 'User Service', maxDelay: 0 },  
5 { utilization: 2.6334, componentName: 'Order DB', maxDelay: 0 },  
6 { utilization: 5.4648, componentName: 'Article DB', maxDelay: 0 },  
7 { utilization: 5.87, componentName: 'User DB', maxDelay: 0.796025 }  
8 ]
```

---

---

**Listing 6.6** Result of performance analysis with average arrival rate of 30 milliseconds

---

```
1 [  
2 { utilization: 0.66, componentName: 'Article Service', maxDelay: 0 },  
3 { utilization: 1.254, componentName: 'Shopping Service', maxDelay: 0 },  
4 { utilization: 1.96667, componentName: 'User Service', maxDelay: 0 },  
5 { utilization: 4.389, componentName: 'Order DB', maxDelay: 0 },  
6 { utilization: 9.108, componentName: 'Article DB', maxDelay: 0 },  
7 { utilization: 9.78333, componentName: 'User DB', maxDelay: 281.917 }  
8 ]
```

---

---

**Listing 6.7** Result of performance analysis with average arrival rate of 30 milliseconds and *User DBs* “maxConcurrency” annotation increased to 100

---

```
1 [  
2 { utilization: 0.66, componentName: 'Article Service', maxDelay: 0 },  
3 { utilization: 1.254, componentName: 'Shopping Service', maxDelay: 0 },  
4 { utilization: 1.96667, componentName: 'User Service', maxDelay: 0 },  
5 { utilization: 4.389, componentName: 'Order DB', maxDelay: 0 },  
6 { utilization: 9.108, componentName: 'Article DB', maxDelay: 0 },  
7 { utilization: 9.78333, componentName: 'User DB', maxDelay: 0 }  
8 ]
```

---

---

**Increasing “maxConcurrency” to remove the bottleneck**

To remove the bottleneck, the “maxConcurrency” performance annotation of the bottleneck component can be increased. Further analysis with the same model with the *User DBs* “maxConcurrency” annotation increased to 100 has been conducted to validate the effect of eliminating the bottleneck. The result with an arrival rate of 30 milliseconds can be found in Listing 6.7. It can be seen that no more delay occurs. As the utilization is growing linearly with the number of users entering the system but the delay grows exponentially as soon as a resource is saturated, this result is expected. Listing 6.8 further shows that with each stateful component having a “maxConcurrency” performance annotation of 100, the system can even handle an arrival rate of 4 milliseconds without any delay before the next bottleneck occurs in the form of the *User DB* and *Article DB* at an arrival rate of 3 milliseconds as can be seen in Listing 6.9.

---

**Listing 6.8** Result of performance analysis with average mean arrival rate of 5 milliseconds and all stateful components “maxConcurrency” annotation at 100

---

```
1 [  
2   { utilization: 4.95, componentName: 'Article Service', maxDelay: 0 },  
3   { utilization: 9.405, componentName: 'Shopping Service', maxDelay: 0 },  
4   { utilization: 14.75, componentName: 'User Service', maxDelay: 0 },  
5   { utilization: 32.9175, componentName: 'Order DB', maxDelay: 0 },  
6   { utilization: 68.31, componentName: 'Article DB', maxDelay: 0 },  
7   { utilization: 73.375, componentName: 'User DB', maxDelay: 0 }  
8 ]
```

---

---

**Listing 6.9** Result of performance analysis with average arrival rate of 3 milliseconds and all stateful components “maxConcurrency” annotation at 100

---

```
1 [  
2   { utilization: 6.6, componentName: 'Article Service', maxDelay: 0 },  
3   { utilization: 12.54, componentName: 'Shopping Service', maxDelay: 0 },  
4   { utilization: 19.6667, componentName: 'User Service', maxDelay: 0 },  
5   { utilization: 43.89, componentName: 'Order DB', maxDelay: 0 },  
6   { utilization: 91.08, componentName: 'Article DB', maxDelay: 2.87025 },  
7   { utilization: 97.8333, componentName: 'User DB', maxDelay: 23.5832 }  
8 ]
```

---



## 7 Conclusion and Outlook

In this work, a concept to enable the usage of the state-of-the-art LQN performance model for conducting a performance analysis on Clams models using variable workload attributes has been proposed. The thesis combines the domains of SPE and cloud computing by allowing a cloud architect to annotate their application model with performance annotations to analyze how the system will behave under a specific workload already in the design phase. Analyzing an Clams application model using the system that has been developed as a part of this thesis makes it possible to detect bottlenecks that occur from using components in an architecture model that are not horizontally scalable and thus might break under an increasing workload. The concepts proposed in this work have been implemented in the form of an evaluation service as an extension to the OpenClams environment.

There are drawbacks to the proposed solution that mostly stem from parts of the Clams model not being transformable to the LQN format. For example, LQN models can not have recursive loops between activities which would be necessary to fully support all modeling capabilities of Clams.

### Outlook

Currently, using the evaluation service is a manual task that has to be started by the OpenClams user via its GUI. In the future, the performance analysis could be conducted in an automatic manner, whenever the model changes. For this, the performance annotations need to be defined for all existing components. The cloud architect would also have to initially provide the workload attributes because the dialog when starting the evaluation would no longer exist. Future work could also use the system proposed in this thesis to automatically determine the maximum workload that a performance-annotated Clams model can handle before bottlenecks occur. This could be especially useful for calculating the maximum arrival rate before the application will break after each change to its structure.

Further, alternative components could be recommended that, for example, represent a vertically scaled version of the currently used stateful component that represents a bottleneck. For this to be possible, the OpenClams component model would need to be extended in order for components of the component tree to be able to hold performance information like the maximum possible number of concurrent connections.





## Bibliography

- [Ars04] A. Arsanjani. “Service-oriented modeling and architecture”. In: *IBM developer works* 1 (2004), p. 15 (cit. on p. 19).
- [BBK+16] A. Bergmayr, U. Breitenbücher, O. Kopp, M. Wimmer, G. Kappel, F. Leymann. “From Architecture Modeling to Application Provisioning for the Cloud by Combining UML and TOSCA.” In: *CLOSER (2)*. 2016, pp. 97–108 (cit. on p. 29).
- [BTN+14] A. Bergmayr, J. Troya Castilla, P. Neubauer, M. Wimmer, G. Kappel. “UML-based cloud application modeling with libraries, profiles, and templates”. In: *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)(2014)*, p 56-65. CEUR-WS. 2014 (cit. on p. 29).
- [BW98] A. W. Brown, K. C. Wallnau. “The current state of CBSE”. In: *IEEE software* 15.5 (1998), pp. 37–46 (cit. on p. 19).
- [DMPS17] E. Di Nitto, P. Matthews, D. Petcu, A. Solberg. *Model-driven development and operation of multi-cloud applications: the MODAClouds approach*. Springer Nature, 2017 (cit. on p. 30).
- [FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud computing patterns: fundamentals to design, build, and manage cloud applications*. Springer, 2014 (cit. on pp. 19, 25, 30, 33).
- [FMW+05] G. Franks, P. Maly, M. Woodside, D. C. Petriu, A. Hubbard, M. Mroz. “Layered queueing network solver and simulator user manual”. In: *Dept. of Systems and Computer Engineering, Carleton University (December 2005)* (2005), pp. 15–69 (cit. on pp. 20, 36, 49).
- [Goo] Google Cloud. *Quotas and limits*. URL: <https://cloud.google.com/sql/docs/mysql/quotas> (cit. on p. 26).
- [GP05] G. P. Gu, D. C. Petriu. “From UML to LQN by XML algebra-based model transformations”. In: *Proceedings of the 5th international workshop on Software and performance*. 2005, pp. 99–110 (cit. on p. 37).
- [Isr01] T. A. Israr. “A lightweight technique for extracting software architecture and performance models from traces”. PhD thesis. Carleton University, 2001 (cit. on pp. 23, 37).
- [KR08] H. Koziolk, R. Reussner. “A model transformation from the palladio component model to layered queueing networks”. In: *SPEC International Performance Evaluation Workshop*. Springer. 2008, pp. 58–78 (cit. on pp. 23, 40).
- [Ley09] F. Leymann. “Cloud Computing: The Next Revolution in IT”. In: *Photogrammetric Week ‘09*. Wichmann Verlag, 2009, pp. 3–12 (cit. on p. 19).

- [MG+11] P. Mell, T. Grance, et al. “The NIST definition of cloud computing”. In: (2011) (cit. on p. 29).
- [Mic] Microsoft Azure. *Limits in Azure Database for PostgreSQL - Single Server*. URL: <https://docs.microsoft.com/en-us/azure/postgresql/concepts-limits> (cit. on p. 26).
- [NLT+11] D. K. Nguyen, F. Lelli, Y. Taher, M. Parkin, M. P. Papazoglou, W.-J. van den Heuvel. “Blueprint template support for engineering cloud-based services”. In: *European Conference on a Service-Based Internet*. Springer. 2011, pp. 26–37 (cit. on p. 29).
- [NMMA16] I. Nadareishvili, R. Mitra, M. McLarty, M. Amundsen. *Microservice architecture: aligning principles, practices, and culture*. O’Reilly Media, Inc.", 2016 (cit. on pp. 19, 25).
- [PS02] D. C. Petriu, H. Shen. “Applying the UML performance profile: Graph grammar-based derivation of LQN models from UML specifications”. In: *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer. 2002, pp. 159–177 (cit. on p. 37).
- [PW05] D. B. Petriu, M. Woodside. “Software performance models from system scenarios”. In: *Performance Evaluation* 61.1 (2005), pp. 65–89 (cit. on pp. 23, 37, 40).
- [RS95] J. A. Rolia, K. C. Sevcik. “The method of layers”. In: *IEEE transactions on software engineering* 21.8 (1995), pp. 689–700 (cit. on p. 33).
- [UKM03] S. Uchitel, J. Kramer, J. Magee. “Synthesis of behavioral models from scenarios”. In: *IEEE Transactions on Software Engineering* 29.2 (2003), pp. 99–115 (cit. on pp. 23, 30, 31).
- [UKM04] S. Uchitel, J. Kramer, J. Magee. “Incremental elaboration of scenario-based specifications and behavior models using implied scenarios”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 13.1 (2004), pp. 37–85 (cit. on pp. 23, 30, 31).
- [WFP07] M. Woodside, G. Franks, D. C. Petriu. “The future of software performance engineering”. In: *Future of Software Engineering (FOSE’07)*. IEEE. 2007, pp. 171–187 (cit. on p. 20).
- [Woo89] C. M. Woodside. “Throughput calculation for basic stochastic rendezvous networks”. In: *Performance Evaluation* 9.2 (1989), pp. 143–160 (cit. on p. 33).
- [WR96] C. M. Woodside, G. Raghunath. “General Bypass Architectures for High-Performance Distributed Applications”. In: *Data Communications and their Performance*. Springer, 1996, pp. 51–65 (cit. on p. 33).
- [XOWM05] J. Xu, A. Oufimtsev, M. Woodside, L. Murphy. “Performance modeling and prediction of enterprise JavaBeans with layered queuing network templates”. In: *Proceedings of the 2005 Conference on Specification and Verification of Component-based Systems*. 2005, 5–es.

All links were last followed on the 22th of June 2021.

### **Declaration**

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature