PERFORMANCE MODELING AND CAPACITY PLANNING FOR SAAS APPLICATIONS

PERFORMANCE MODELING AND CAPACITY PLANNING FOR SAAS APPLICATIONS

By NAFISEH VALIZADEH SHIRAN, MASc

A Thesis Submitted to the School of Graduate Studies in Partial Fulfillment of the Requirements for the Degree Master of Applied Science Software Engineering

McMaster University

MASTER OF APPLIED SCIENCE SOFTWARE ENGINEERING (2025)

Hamilton, Ontario, Canada (Computing and Software)

TITLE:	PERFORMANCE MODELING AND CAPACITY PLANNING FOR SAAS APPLICATIONS
AUTHOR:	Nafiseh Valizadeh Shiran MASc (Master of Applied Science Software Engineering), McMaster University, Hamilton, Canada
SUPERVISORS:	Professor Douglas Down Professor Richard Paige
NUMBER OF PAGES:	x, 62

Lay Abstract

This research aims to simplify the capacity provisioning of cloud-based applications by reducing reliance on extensive lab-based testing and offering more time and cost effective alternatives. We develop a simulation framework integrated with an analytical solution to predict system performance. These methods are validated using empirical test data, demonstrating their effectiveness in assisting system architects to ensure scalability and performance of their services are maintained within acceptable quality of service agreements.

Abstract

Providing quality services in cloud-based systems is a critical factor. SaaS based applications typically experience dynamic workloads which compels system architects and providers to look for ways to keep their application services scalable, but finding the right number of services can be a cumbersome task if they solely rely on testing and maintaining the application in lab environments. This is primarily due to the significant time and costs involved in setting up the system. In this thesis, we propose a lightweight method to perform capacity planning of the applications. This approach combines an analytical tool that models the system as a closed network of queues and utilizes a numerically stable algorithm, SMVA, to approximate performance metrics, and a simulation framework developed to capture more intricacies of our underlying system environment and platform. We validate the proposed methods on a microservices app deployed on a Kubernetes cluster that captures key metrics like throughput, response time, and pod CPU utilization. The results show acceptable agreement among the SMVA predictions, simulation outputs, and empirical observations, therefore confirming the effectiveness of our approach.

To the resilient students of my homeland, whose pursuit of education continues despite the weight of conflict.

Acknowledgements

I would like to begin by gratefully acknowledging the financial support I received from the Natural Sciences and Engineering Research Council of Canada (NSERC) and Cubic Transportation Systems. This work would not have been possible without their generous support.

I would also like to express my deepest gratitude to my academic supervisor, Dr. Douglas Down, for his invaluable guidance, insight, and continuous encouragement throughout my research. His mentorship has been a key part of my academic journey, and I am truly thankful for all the support and advice he has given me.

I also want to thank my co-supervisor, Dr. Richard Paige, for his support during my time in the program.

I am grateful to the faculty and staff at McMaster University for their support throughout my studies.

Most importantly, I wish to thank my family. Their constant presence, patience, and belief in me made all the difference, especially during the most challenging moments. I'm deeply thankful for everything they've done to help me get here.

Table of Contents

List of Figures

List of Tables

Declaration of Academic Achievement

The following is a declaration that the research represented in this thesis was completed by Ms. Nafiseh Valizadeh Shiran and acknowledges the contributions of Dr. Douglas Down and Dr. Richard Paige.

Together with Dr. Douglas Down and Dr. Richard Paige, Ms. Nafiseh Valizadeh Shiran contributed to the inception of the study and the overall study design. She was primarily responsible for the implementation of the performance models, development of the simulation and algorithms, execution of experiments, and the writing of the manuscript. Dr. Douglas Down and Dr. Richard Paige contributed to the inception and design of the study, reviewed the manuscript, and provided ongoing guidance and support throughout all stages of this thesis. This thesis contains no material that has been submitted or published previously, in whole or in part, for the award of any other academic degree or diploma, except where otherwise indicated. This thesis is entirely the original work of Ms. Nafiseh Valizadeh Shiran.

Chapter 1

Introduction

Cloud computing has fundamentally changed the way resources and services are provided with its various types of service provisioning models such as infrastructure as a service (IaaS), software as a service (SaaS), and platform as a service (PaaS). It provides businesses a wide range of solutions that they can adopt. It allows businesses to dynamically scale and increase their agility while also helping them to manage costs based on their consumption [19].

To maintain customer demands and ensure service quality, cloud providers face specific challenges in efficiently managing their resources, both hardware and software. These challenges are not limited to public cloud providers; they also affect small enterprises that adopt cloud computing paradigms for provisioning their private on-premise cloud infrastructure.

In this work our main focus is on the software delivery model that provides applications directly to end-users. We focus on the scalability challenge in SaaS applications, because of their highly variant load and unpredictable changes, although this challenge is also important across all cloud models. Microservices architecture is a pattern widely adopted to facilitate granular resource control and bottleneck identification. For larger-scale applications, this architecture is used with containerization approaches. To manage complexities inherent in such environments, orchestration platforms like Kubernetes are utilized. Kubernetes facilitates fast and controlled deployment of services while providing scaling functionality as a key feature that smooths the need for both application agility and availability.

Effective scalability requires workload characterization, user demand patterns, resource availability, and Quality of Service (QoS) constraints. Therefore, achieving efficient service delivery requires proactive system profiling and capacity planning.

To effectively utilize Kubernetes and meet QoS constraints, such as maintaining response times and sufficient throughput, and also to avoid the complexities of deploying and testing actual application services (which is a costly and time-consuming task), we can benefit from analytical and simulation models. In this work, we use queueing theory, a subset of stochastic modeling, which characterizes the inter-arrival times of requests as well as their processing demands. We use principles of queueing theory to represent our services as networks of queues. To represent changes in the state of the system, which consists of services and incoming requests, we utilize Continuous-Time Markov Chains (CTMC). Then, to estimate important performance metrics such as response times and throughput, we apply an approach called Stable Mean Value Analysis (SMVA). Additionally, we develop a capacity planning tool that provides us with a first cut of the service requirements that enables more accurate resource allocation decisions in the design phase.

Analytical models provide theoretical insights and an initial performance evaluation of systems. However, since they often require simplifying assumptions regarding arrival processes, service demand patterns, and their interactions, they might not fully capture the intricacies present in real-world systems. Thus, we have decided to also utilize simulation models. The simulation framework developed in this work enables us to capture detailed interactions between the services as well as the complexities in the underlying container orchestration tool, Kubernetes.

As our validation scenario, we use the Spring PetClinic Cloud microservices application, which is deployed on a Kubernetes cluster. This application provides a realistic example to model its behavior as a closed network of queues. We leverage this setup to directly compare and validate the performance predictions from our analytical and simulation models. The open-source project is available on GitHub. Figure 1.1 provides a high-level overview of the microservices structure and illustrates the flow of user requests through the application.

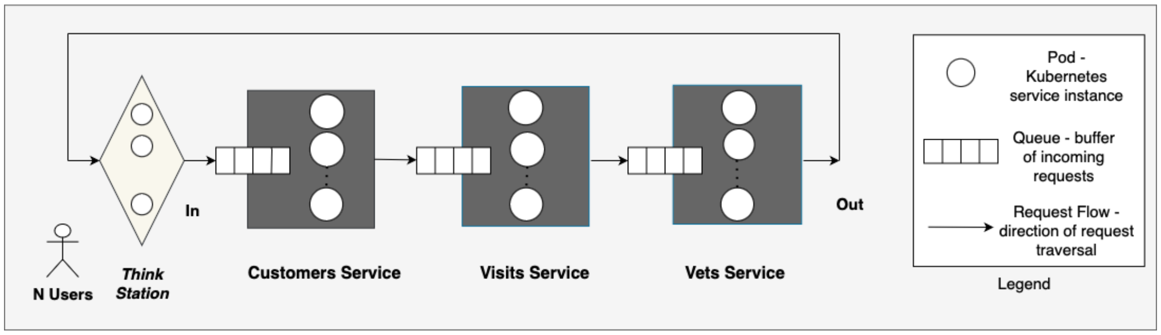
N Users
Think Station
In
Customers Service
Visits Service
Vets Service
Out
Pod - Kubernetes service instance
Queue - buffer of incoming requests
Request Flow - direction of request traversal
Legend

Figure 1.1: High-level overview of the microservices structure and flow of user requests in the PetClinic application.

The remainder of this thesis is organized as follows. Chapter 2 provides a comprehensive literature review, discussing existing analytical and simulation approaches relevant to performance modeling and capacity planning of computer systems, including cloud-based services. Thereafter, in Chapter 3, we describe our methodology, including our analytical approach based on queueing theory, solving the Continuous-Time Markov Chain for our application, the use of the SMVA algorithm to derive performance metrics, and our discrete-event simulation framework designed to overcome the limitations of analytical modeling. Chapter 4 covers implementation aspects, explaining how we set up our Kubernetes cluster environment and deployed and configured the Spring PetClinic application. Chapter 5 presents our experimental results, comparing performance measurements obtained from running the Spring PetClinic application on a production-ready Kubernetes cluster against predictions from the SMVA approach and simulation (DES) model. Lastly, in Chapter 6, we draw conclusions based on our methodology's results and offer recommendations for future research.

Chapter 2

Literature Review

This chapter reviews existing research on performance modeling of microservices and cloud-based services. The literature is categorized based on methodological grouping: Analytical Queueing Models, Simulation Approaches, and Hybrid and Empirical Validation Studies.

2.1 Analytical Queueing Models

Queueing models form the essential foundation which supports both performance prediction and resource allocation methods. Research studies [7, 25] utilize simple queueing models ($M/M/1$, $M/M/1/C$, and $M/M/k/k$) for multitier systems for predicting response time and throughput performance metrics. The models demonstrate benefits but their validation depends only on simulator-based tests using Java Modeling Tools (JMT) and CloudSim without verification from actual system implementations.

The research by Carvalho et al. [2] uses queueing theory to design a multi-class capacity planning model for IaaS platforms which includes quota-based admission control and service-level objectives (SLOs). The approach optimizes CPU usage effectively but its primary focus is on CPU resource modeling. The approach does not capture the detailed microservice-level interactions, job flows and load-dependent scaling behaviors which happen in applications running on cloud infrastructures.

Kouki and Ledoux [17] discuss the challenge of resource allocation for multi-tier SaaS architectures in which multiple user classes are categorized based on their cost and performance requirements. The SaaS system is represented as a closed queueing network where each tier is modeled as a $M/M/c/k$ queue. They compute metrics like request latency and abandonment rates by using an extended Mean Value Analysis (MVA) algorithm. A utility function uses these metrics to balance application cost and dependability. A capacity planning tool is then implemented to suggest the optimal configuration that adheres to the SLAs. Similar to other studies so far, the validation of this work also does not perform empirical assessments, relying on simulation results.

The paper by Casale [3] presents a solution to the numerical instability of load-dependent MVA. This extension of MVA, is known to exhibit numerical instability, particularly when processing rates depend on the load; therefore, they used Buzen's convolution algorithm to address this problem. This method has the same complexity of $\mathcal{O}(MN^2)$ as the load-dependent MVA, but addresses the instability issue by avoiding probability computations. However, it is still not computationally efficient for systems with a large number of users and servers. On the other hand, the Stable Mean Value Analysis (SMVA) approach [28], which also addresses the same issue, offers a straightforward mapping between the real metrics of the system and the theoretical model. However, it is important to note that SMVA is an approximate algorithm rather than an exact solution.

2.2 Simulation Approaches

Simulation tools are beneficial in terms of capturing complex properties of software systems and interactions between their components and could be used as complementary methods for the performance evaluation of such systems [14, 20]. The simulation framework µqsim [29] simulates multi-tier microservices and captures metrics such as average and tail latencies. It targets concurrency and thread-level resource management. However, it does not simulate the interaction of microservices, the request flow, and routing and scaling features of applications running on the cloud infrastructure - features that we desire for our analysis.

Tian et al. [26] provide a comparative survey of existing simulation tools. While these tools provide insights on event-driven execution and hierarchical modeling of our framework, many of them target traditional VM structures rather than simulating the behavior and characteristics of applications deployed on orchestration environments.

The PACE framework proposed by Chouliaras and Sotiriadis [5] combines reactive and proactive mechanisms for resource allocation and workload surges. The framework is fast and real-time for scaling decisions, but it does not cover deeper queueing dynamics of services. The work represented in this thesis provides more detailed insight into how communication patterns influence the performance of services and the overall system.

Pinciroli et al. [21] discuss simulating microservices based on design patterns and how each of these patterns could influence the performance of the application. They further evaluate the patterns with a JMX-based tool. Although they have proposed extending their work by using Markov chains, their current contribution remains on simulation experiments and not comparing with empirical tests.

2.3 Hybrid and Empirical Validation Studies

Hybrid methods, combining analytical models with empirical validations, offer practical insights into the performance of microservices.

Khazaei et al. [16] designed an analytical model using Continuous-time Markov chains that focuses on the provisioning dynamics of containers and virtual machines for microservice platforms. Their study provides valuable insight on reducing the time and cost of infrastructure provisioning and elasticity through their systematic capacity planning algorithm.

Han et al. [10] introduce a heuristic-based method for efficient placement of microservices on Kubernetes clusters. They use empirical test data for this decision making based on metrics such as response time and resource usage. However, since their method lacks theoretical analysis for optimal resource placement, it may be limited for varying and heavy workloads.

Similarly, Jindal et al. [15] utilize Terminus to perform sandboxing of microservices. In their approach, they first isolate each microservice and perform load tests, employing regression analysis on the data to find the maximum request rate each can handle without violating the SLOs. Their approach does not consider the impact of inter-service communication on the system performance.

Chen et al. [4] focus mainly on exploring the benefits of breaking down monolithic applications to microservices by analytical decomposition based on simple $M/M/1$ and $M/D/1$ queueing models. In their study, they try to quantify the impact of this decomposition on latency and resource usage by running simple scenarios. However, they oversimplify important performance issues, such as request routing and load-dependent scaling behaviors.

Furman and Diamant [8] utilize a modified Offered Load Function to perform capacity planning for cloud services that experience demand fluctuations. Their method suggests retrial intervals for user requests that could shift the load during peak times to off-peak periods. While their work is effective for macro-level capacity planning, it can be limited in its applicability as it simplifies operational features like internal queueing and load balancing.

The study by Urgaonkar et al. [27] presents a modeling approach for multi-tier internet applications. The model includes complex features of applications such as concurrency limits, caching, and load distribution across multiple service instances. However, their method relies on standard iterative MVA computations, which inherently may cause numerical instabilities, particularly at larger scales. This thesis uses Stable Mean Value Analysis (SMVA) [28] which addresses the issue through a decomposition strategy, by isolating load-dependent service behavior into simpler calculations. The detailed operational insights provided in [27], such as service concurrency management and caching effects, represent beneficial enhancements that could potentially enrich MVA-based approaches.

2.4 Critical Summary and Research Gap

The reviewed literature highlights methodological advancements alongside notable limitations. Analytical queueing models [2, 3, 6, 7, 17, 23, 25] frequently lack empirical validation against real-world scenarios or don't fully capture the detailed complexities of request behaviors within applications environments. Simulation-based studies [5, 21, 26, 29] while effective at modeling runtime behaviors such as intra-service concurrency, often simplify or overlook the detailed interactions and orchestration dynamics inherent in microservice deployments.

Hybrid methods [4, 8, 10, 15, 16, 27] address operational realism, but typically focus narrowly (provisioning delays, isolated services, cyclical loads) or omit key complexities such as service interactions, internal queueing, or realistic concurrency management.

In contrast, this thesis integrates

- An analytical model (closed Jackson network), analyzed using the Stable Mean Value Analysis algorithm, is utilized for detailed request-level performance predictions and service interaction complexities in microservice applications.
- Empirical validation through application load-testing data, to confirm analytical and simulation predictions against real deployments.
- A complementary discrete-event simulation framework modeling microservice interactions, request flows, and routing behaviors influenced by the details of the underlying container orchestration engine.
- A capacity planning tool that provides practical recommendations based on

SLA-driven resource allocation validated using analytical, simulation, and empirical methods.

Consequently, this thesis addresses the methodological gaps identified in analytical, simulation, and hybrid studies, and offers empirically validated insights that are aimed at performance modeling and capacity planning in microservice-based applications operating in container orchestration environments.

Chapter 3

Methodology

This chapter explores analytical methods that facilitate the performance prediction of SaaS-based applications, specifically microservices architectures. We first discuss how we could model our system as a Jackson network of queues. Building upon this, we show that the system state can be represented through a Continuous-Time Markov Chain (CTMC) which we explain in detail along with its characteristics and methods for calculating steady-state probabilities. This discussion on CTMCs is followed by an introduction to the Jackson Product Form theorem, which is used to calculate steady-state probabilities. Finally, we suggest the Stable Mean Value Analysis (SMVA) method as an extension of traditional mean value analysis (MVA). We explain our decision to use SMVA, emphasizing its advantage to estimate mean response times and throughput.

3.1 Queueing Networks

In the context of queueing theory, we can think of microservices as individual nodes, each with its own processing demand. If the node/server is idle, the arriving request can be processed immediately. However, if the request finds all the servers busy, it should wait in the queue until one server becomes available. These nodes are often interconnected such that the output from one service is the input to another; thus their interactions can be represented as a network of queues.

Jackson networks are queueing networks characterized by exponential service times and probabilistic routing between multiple nodes. A Jackson network consists of M nodes, each serving requests in First-Come-First-Served (FCFS) order, with processing rates μ_i for each node i. Arrivals in the network may originate from outside the system without being influenced by how requests leave the system, or they can originate internally as requests transition between nodes determined by routing probabilities P_{ij} which represent the probability of a job moving from server i to server j. Jackson networks can be classified into the following categories:

- **Open Networks:** There are external arrivals and departures.
- **Closed Networks:** A fixed number of requests circulate internally without external arrivals or departures.

In our testing scenarios, users, after thinking for a period of time, submit a request for the sequence of microservices. Once the request is completed, the entire process is repeated. Since there are a fixed number of users in the system without external arrivals or departures and due to predefined routing probabilities between microservices, the closed Jackson network is a suitable representation. To analyze and solve such closed networks, we represent the system using a Continuous-time Markov chain (CTMC).

3.1.1 Continuous-Time Markov Chain

A Continuous-Time Markov Chain (CTMC) is a stochastic process characterized by state transitions that occur according to exponentially distributed random variables over continuous time intervals. In the context of our scenario, the CTMC provides a representation of the state of the system by reflecting the distribution of requests among the nodes.

Formally, we define our CTMC state as a vector:

$$\mathbf{n} = (n_1, n_2, \ldots, n_M),$$

where n_i indicates the number of requests present at the i-th node. The number of users in the think node is given by $n_{M+1} = N - \sum_{i=1}^{M} n_i$, where N is the total number of users. Transitions between these states occur at the rates of the preceding nodes. When a node completes processing a request, the request moves to the next node with a pre-defined routing probability. Figure 3.1 illustrates the CTMC transitions for a scenario with two tasks in the system where each microservice only has one pod. To solve the CTMC and find steady-state probabilities, we need to formulate global balance equations for each state. The global balance equations ensure that the total rate entering a state is equal to the total rate leaving the state. In the next section, we will see that the steady-state probabilities of the CTMC for Jackson networks can be expressed in product form.

Having formally defined the CTMC as a stochastic process describing system states and state transitions, we can now apply it to our network of queues. Table 3.1 is summary of all notations used in the preceding and upcoming sections.

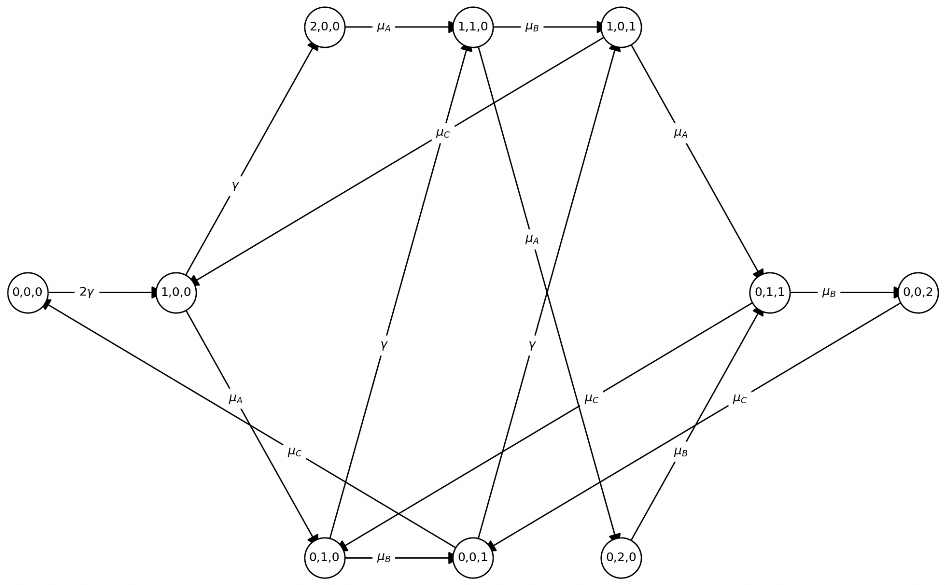

Figure 3.1: Continuous-Time Markov Chain (CTMC) state diagram illustrating transitions and rates. μ_A, μ_B, and μ_C are the processing rates at Customers, Visits, and Vets service nodes, respectively.

3.2 Product Form Solution

To find the steady-state probabilities of our microservices system modeled as a Continuous-Time Markov Chain (CTMC), we use the Gordon-Newell theorem [9], which builds upon Jackson's theorem [12, 13]. These theorems assume exponential service times at each node and unlimited queue capacities.

Notation	**Description**
M	Total number of microservice nodes
N	Total number of users in the network
n_i	Number of users at microservice node i
μ_i	Processing rate (service rate) at node i
γ	User request generation rate
λ_i[1]	Arrival rate at node i
s_i	Number of instances (servers/pods) at node i
P_{ij}	Routing probability from node i to node j
ρ_i	Utilization of node i
$\pi_{\mathbf{n}}$	Steady-state probability of system being in state $\mathbf{n}$
C	Normalizing constant
Z	Mean think time at the think node
X_n	Throughput for population n
$R_m(n)$	Response time at node m with population n
$D_m(n)$	Service demand at node m for n requests
$\overline{N}_m$	Concurrency level at node m
$Q^o_m(n)$	Total occupancy at node m at population n
$Q^e_m(n)$	Estimated average number of users at delay center
$Q_{a,m}(n)$	Front-server queue length at node m

Table 3.1: Summary of notations.

In our system, we consider a closed Jackson network consisting of M microservice

[1]Represents a relative arrival rate solution to the balance equations.

nodes, each operating as a load-dependent queue with exponential processing times, represented as $\cdot/M/s_i$. Additionally, we include a think node (node $M+1$) modeled as an infinite-server queue ($\cdot/M/\infty$), labeled as node $M+1$. The total number of users in the system is N, which satisfies:

$$\sum_{i=1}^{M+1} n_i = N.$$

The service rate at each node i is defined in a piecewise manner as:

$$\mu_i(n_i) = \begin{cases} \min(n_i, s_i)\,\mu_i, & i = 1, \ldots, M \\ n_i\,\mu_i, & i = M+1 \end{cases}$$

The generalized local balance equations for each state $(n_1, \ldots, n_{M+1})$, are given by:

$$\pi(n_1, \ldots, n_{M+1})\,\mu_i(n_i) = \sum_{j=1}^{M+1} \pi(n_1, \ldots, n_j + 1, \ldots, n_i - 1, \ldots, n_{M+1})\,\mu_j(n_j + 1)\,P_{ji},$$

where $n_i > 0$ and $n_j < N$.

Due to the computational cost of explicitly solving these equations for large M and N, with a time complexity of $\mathcal{O}(N^M)$ we leverage the established results from the Gordon–Newell theorem [9], which provides a closed-form solution for the steady-state probabilities.

From these local balance equations, we derive the simultaneous rate equations for each node:

$$\lambda_i = \sum_{j=1}^{M+1} \lambda_j P_{ji}, \quad \text{with} \quad \sum_{j=1}^{M+1} P_{ij} = 1.$$

This set of equations does not yield a unique solution. We may therefore select an arbitrary solution by setting one of the λ_i equal to 1, and then use the normalization constant to determine the actual arrival rates λ_i. We define

$$\rho_i = \frac{\lambda_i}{s_i \mu_i}, \quad i = 1, \ldots, M+1.$$

If the values of λ_i were the true arrival rates, then ρ_i would be the utilization of node i. While the true arrival rates are unknown, the quantities ρ_i are sufficient to calculate the steady-state distribution.

Using these assumptions and the Gordon-Newell theorem, the steady-state probability of the system for state $\mathbf{n} = (n_1, \ldots, n_M, n_{M+1})$ is given by the Jackson product form:

$$\pi_{\mathbf{n}} = C \left(\prod_{i=1}^{M} \frac{(s_i \rho_i)^{n_i}}{\beta_i(n_i)} \right) \frac{(s_{M+1} \rho_{M+1})^{n_{M+1}}}{n_{M+1}!},$$

where

$$\beta_i(n_i) = \begin{cases} n_i!, & n_i \leq s_i, \\ s_i! \, (s_i)^{n_i - s_i}, & n_i > s_i \end{cases}$$

The normalization constant C is determined by the condition:

$$\sum_{\mathbf{n}} \pi_{\mathbf{n}} = 1.$$

The product form solution provides us with the steady-state probability for all possible states in our CTMC. However, computing the normalizing constant, especially for larger N or multiple load-dependent servers, is demanding. With this in mind, since our goal is to find approximate values for metrics such as response times and throughput, we turn to a less computationally expensive approach called Mean Value Analysis (MVA).

3.2.1 PetClinic Application as a Closed Jackson Network

The characteristics of the PetClinic application, such as user interactions and microservice communication, have led us to model the system as a closed Jackson network of queues. Users are continuously circulating through the system. Each request initiated by a user proceeds through a sequence of services, including customer service, followed by visit service, and then vet service, before reaching the think node. After spending some time in the think node, each user repeats the same cycle again. The system maintains a steady user population because new requests neither enter nor leave the system.

In addition to this cyclic behavior, microservices use predefined routing probabilities to interact with each other. The output of one microservice serves as the input of another, with transitions occurring based on routing probabilities P_{ij} and processing rates $\mu_i(n_i)$. For example, consider a transition from state $(1, 0, 0)$ to state $(0, 1, 0)$, which indicates a request that transitions from customer service to visit service. This transition occurs at rate μ_{customer}, reflecting the completion of a request in customer service and the route to visit service.

As the system scales, the complexity of this model increases substantially with the number of microservices M and users N. Specifically, determining the normalization constant C used by the product form solution, involves adding the probabilities of all possible states of the system, given by $\binom{N+M}{M}$.

Figure 1.1 illustrates the conceptual model of this microservices application system. In this figure, each microservice is a node that consists of one or more pods, with arrows indicating the direction of request flows between microservices.

3.3 Stable Mean Value Analysis (SMVA)

Stable Mean Value Analysis (SMVA) [28] is an extension of the classical Mean Value Analysis Method to handle load-dependent queues in a closed network, while avoiding numerical instability. This issue happens in MVA because of its iterative nature in computing state probabilities for load dependent queues which leads to cumulative rounding errors. These rounding errors can lead to significant errors in mean response time calculations, such as negative values.

SMVA prevents this by splitting each load-dependent node into a front server (with constant demand $D_m(\overline{N}_m)$) and a delay center, which captures the leftover portion via

$$D_m^{(d)}(n) \;=\; \begin{cases} n\,D_m(n) \;-\; D_m\big(\overline{N}_m\big), & \text{if } n < \overline{N}_m, \\[2ex] \big(\overline{N}_m - 1\big)\,D_m\big(\overline{N}_m\big), & \text{if } n \geq \overline{N}_m. \end{cases}$$

For a single class with total population N, SMVA proceeds from $n = 1$ up to $n = N$. Let $Q_m^o(n)$ be the total number of requests at node m at population n, and let $Q_m^e(n)$ be the estimated number of requests at the delay node. We set $Q_m^o(0) = 0$

initially. Then, for each n:

1. **Estimate number of requests at delay-center**

   In this step, we use the Bard-Schweizer [1, 22] approximation to estimate a delay-center's mean number of requests at the mth queue. If $n = 1$, we simply set $Q^e_m(1) = 1$. For $n > 1$, we have:

$$Q^e_m(n) \;=\; \frac{n}{n-1} \;\times\; Q^o_m(n-1).$$

2. **Compute front-server response time**

   Suppose node m's front demand is $D_m\big(\overline{N}_m\big)$ and we treat it as an M/M/1 queue. Then the front-server mean response time $R_{q,m}(n)$ is

$$R_{q,m}(n) \;=\; D_m\big(\overline{N}_m\big)\,\big(1 + Q_{a,m}(n-1)\big),$$

   where $Q_{a,m}(n-1)$ is the front-server mean queue length from the previous step.

3. **Compute delay-center response time**

   Using $Q^e_m(n)$ as the estimated number of requests at the delay-center, we take $\lceil Q^e_m(n)\rceil$ and insert it into the delay center demand function $D^{(d)}_m$:

$$R_{d,m}(n) \;=\; D^{(d)}_m\big(\,\lceil Q^e_m(n)\rceil\big).$$

4. **Sum node response times**

   After calculating the front and delay center's mean response times, the node total mean response time is

$$R_m(n) \;=\; R_{q,m}(n) + R_{d,m}(n).$$

5. **Compute throughput**

   Let Z be the mean think time. Then the overall throughput for system population n is

$$X_n = \frac{n}{Z + \sum_m R_m(n)}.$$

6. **Update front-server queue length**

   We update the front-server mean queue length at node m for population n

$$Q_{a,m}(n) \;=\; X_n \;\times R_{q,m}(n).$$

7. **Update total node queue length**

   Finally, the total mean number of requests at node m is

$$Q^o_m(n) \;=\; X_n \;\times \Big[R_{q,m}(n) \;+\; R_{d,m}(n)\Big].$$

By following these seven steps for $n = N$ iterations, we get estimates of X_N (throughput), $Q^o_m(N)$ (total mean queue length), $Q_{a,m}(N)$ (mean queue length at front queue), and $R_{q,m}(N)$, $R_{d,m}(N)$ (front and delay mean response times). These steps comprise the SMVA approach from [28].

3.4 Algorithm Implementation

The following is a Python implementation of the SMVA approach. It takes as inputs service demands, number of users, N, and mean think time to calculate the performance metrics, throughput and response times of each service. The algorithm iteratively computes the approximate mean queue length and mean response time for each service node following the steps discussed above.

Algorithm 1 Stable Mean Value Analysis (SMVA)

```
1: **Input:** Service demands $D_m(n)$, population $N$, think time $Z$
2: **Initialize:** $Q^o_m(0) \leftarrow 0$ for all $m$
3: **for** $n = 1$ to $N$ **do**
4:     **for** each node $m$ **do**
5:         **if** $n = 1$ **then**
6:             $Q^e_m(1) \leftarrow 1$
7:         **else**
8:             $Q^e_m(n) \leftarrow \frac{n}{n-1} \cdot Q^o_m(n-1)$
9:         $R_{q,m}(n) \leftarrow D_m(\overline{N}_m) \cdot (1 + Q_{a,m}(n-1))$
10:        $R_{d,m}(n) \leftarrow D^{(d)}_m(\lceil Q^e_m(n) \rceil)$
11:        $R_m(n) \leftarrow R_{q,m}(n) + R_{d,m}(n)$
12:    $X_n \leftarrow \frac{n}{Z+\sum_m R_m(n)}$
13:    **for** each node $m$ **do**
14:        $Q_{a,m}(n) \leftarrow X_n \cdot R_{q,m}(n)$
15:        $Q^o_m(n) \leftarrow X_n \cdot [R_{q,m}(n) + R_{d,m}(n)]$
16: **Return:** $X_N$, $Q^o_m(N)$, $R_m(N)$
```

The Stable Mean Value Analysis method that we discussed above is a fast and efficient way to estimate mean performance values such as queue time and response time with time complexity $\mathcal{O}(NM)$. However, product form conditions are assumed while using this method. Real-world examples of applications on the other hand have more complexities which include dynamic routing, concurrency constraints, and non-exponential processing time distributions. Therefore, to address some of these practical challenges and gain more insight into the performance behavior of such systems, we complement our analytical approach with a discrete event simulation approach, described in the following section.

3.5 Discrete-Event Simulation of Microservices

As discussed earlier, analytical methods are powerful tools for characterizing the performance of software systems. However, they rely on simplifying assumptions and fail to capture certain complexities such as queue disciplines and processing policies. Therefore, we have to rely on other solutions as well to complement their shortcomings. Discrete-Event Simulation (DES) is a widely used method that enables us to model detailed system interactions. Discrete-Event Simulation allows system states to evolve through distinct events which occur at specific points during a given simulation period. The DES model includes representations of entities (requests or tasks), resources (processors and queues) and events (arrivals, departures and completions).

The main components of a DES model include:

- **Entities:** These represent the active components in the system, such as requests, customers, or, in our case, user requests and microservice pods.

- **Events:** Any occurrence that changes the state of the system. For instance a request arriving to the system or completing its processing.
- **Queues:** Buffers where entities wait for resources if they are busy. In a software system, this might be the waiting line for a particular service.
- **Resources:** The processing units that serve the entities. In this case, pods serve as resources with a certain capacity for concurrent requests.
- **Simulation Clock:** A global simulation time that advances from one event to the next, rather than proceeding continuously.

3.5.1 Simulation Framework and System Components

We have developed a discrete-event simulation (DES) framework to model the behavior of microservice-based applications running on a Kubernetes cluster. The framework is implemented using `SimPy` library in Python, that lets us implement request arrivals, resource contention, request routing, queueing, and concurrency limits. The simulation architecture includes the following main components:

- **Nodes:** The nodes in the Kubernetes cluster represent virtual machines (VMs) or physical hosts. Nodes act as the hosts for running containerized microservices within the pods. A node can contain multiple pods, each corresponding to specific microservices. Nodes are created based on the configuration given by the user that specifies available services and number of pods per service [2].

[2]In this section, the terms tasks and requests are used interchangeably to refer to individual HTTP requests initiated by users. Similarly, the term services may refer either to microservices or Kubernetes Service objects.

- **Pods:** Each pod represents the atomic processing unit in a Kubernetes cluster. Pods are characterized by their processing rates and concurrency limits which differ for each service. The concurrency capacity is managed by `SimPy Resource` that limits the number of simultaneous tasks that could be processed. When a task arrives at the system, it gets assigned to a pod to be processed based on a processing rate and then the simulation clock advances using `env.timeout()`. Subsequently, the task's processing duration, queued time, response time, and exit time from the pod will be updated.

- **Tasks:** Tasks represent individual HTTP requests sent by users to interact with the microservice application. Each task has a predefined type (e.g., customers request, visits request, vets request) that determines the sequence of services it must traverse. When a task enters the system, it is routed through the specified microservices defined in the sequence. At each microservice, it may experience queueing delays, wait for available resources, and undergo processing before proceeding to subsequent services. Throughout this process, tasks gather performance metrics that are used to evaluate system performance under various conditions.

- **Services:** Each microservice represents an application-level functionality (e.g., customer management, visits management, veterinarian search) that runs within each pod.

- **Task Generator:** Users are simulated as concurrent and independent entities continuously sending requests in a closed-loop pattern. Each simulated user:

  1. Creates tasks based on the defined task types and service sequences.

2. Sends tasks to the system and waits until their completion, after which it records the task's performance metrics such as response time and queue time.
3. Pauses for a randomized think time, before starting the next iteration.

- **Controller:** The controller acts as the central logic handling the routing and execution of tasks. For each incoming task, the controller performs the following responsibilities:

  1. Identifies nodes which have the requested service types.
  2. Selects the node which currently has the shortest queue length across its pods having the relevant service.
  3. Selects the pod within this node that currently has the least load, defined by the sum of the number of active and queued tasks.
  4. Manages task queueing if no pod concurrency slots are immediately available.
  5. Initiates the processing once a pod is available.
  6. Routes the task to the next service after the processing in the current service completes and repeats steps 1 to 5.

Figure 3.2 shows a high-level view of the system. This figure illustrates the flow of requests in the system and the decision making for assigning a resource for the incoming requests.

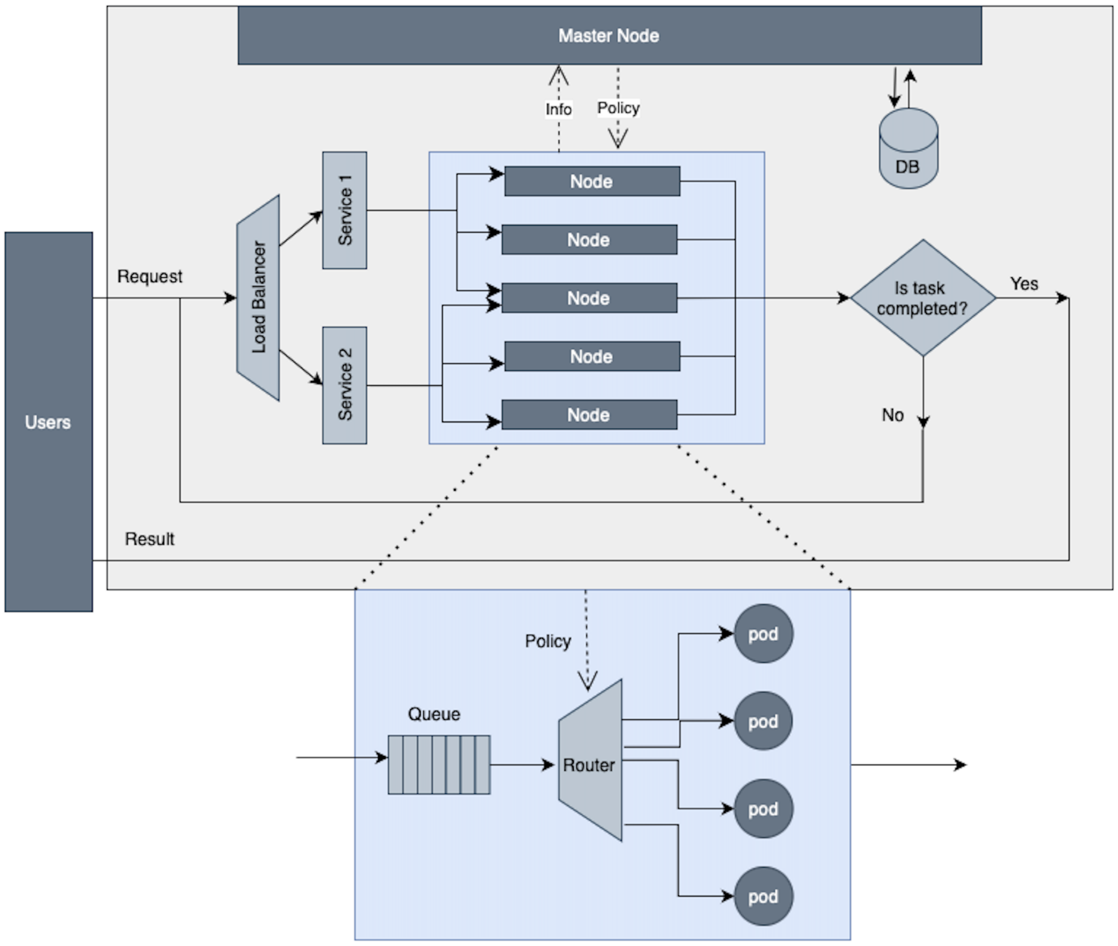

Figure 3.2: High-level overview of task flow and resource selection in the discrete-event simulation model

The detailed operational flow of tasks through this framework will be described in the next subsection.

3.5.2 Event Flow

In our DES, the Simulation Clock starts at time zero. The flow of events is as follows:

1. **Task Arrival:** The `TaskGenerator` generates a new `Task` at time intervals.

This event updates the simulation clock to the arrival time.

2. **Routing and queueing:** The `Controller` identifies a suitable node and pod. The task creates a resource request (`pod.resource.request()`).

3. **Service Start:** Once the task acquires the resource, it begins processing. The task enqueue time is recorded to measure the queue time.

4. **Service Completion:** After a processing delay (`env.timeout(service_time)`), the task completes service on the current pod. This triggers an event to release the resource and log the task's queue and processing times.

5. **Next Service or Exit:** If the task's service chain has additional microservices to visit, it repeats the routing and queueing steps. Otherwise, it exits the system, and its total response time is finalized.

For all entities within this simulation, the simulation clock operates at a global level and it advances through its priority queue for every event in the described steps, which maintains the correct sequence of queue durations.

Chapter 4

Implementation

4.1 Overview of the System Architecture

The application used for this research is the Spring Petclinic Cloud [24], which is a modified version of the original Spring Petclinic microservices. For the purpose of this research that needed a containerized microservices application, this version matched the requirements, as it is specifically designed for deployment in cloud environments using Kubernetes. This makes it particularly suitable for validating the DES approach that we discussed earlier. The purpose of this chapter is to see how to assess the performance of a real-world application running on a cloud infrastructure. The setup of the environment before deploying the application requires at least one virtual machine from a cloud provider on which to set up the Kubernetes cluster. The cluster consists of two nodes. After preparing the environment, we deployed the application and the pods distributed on both nodes. The number of microservice pods can be changed over the course of the service deployment. To be able to monitor the performance results of application load tests with various load types, a monitoring stack is added to the cluster. It constantly monitors the health and status of all objects in the cluster, with dashboards to query our target metrics such as response times to http requests, thread count for each microservice pod, and microservice/pod throughput.

4.2 Kubernetes Cluster Setup

4.2.1 Cluster Configuration

The Kubernetes cluster used in this project is deployed with **Kubespray** which is a collection of Ansible playbooks that automate the configuration and deployment of the cluster within the nodes. The first step to configure **Kubespray** [18] is to clone the official repository and then create an inventory file `inventory/mycluster` to define the node roles in the cluster. Appendix A.1 shows the format of the inventory file to define Node 1 as the control plane or the master node and Node 2 as the worker node.

After configuring the inventory, the desired versions and plugins are specified in the group variables. For this environment, **Kubernetes v1.29.10** is used (a stable release), the **Calico** plugin is used to provide networking and network policies, **containerd** serves as the container runtime (because docker is deprecated), and **local path** is chosen as the default StorageClass to handle local volume provisioning on nodes.

After setting all the required parameters for cluster setup, KubeSpray is run against the specified inventory using:

```
ansible-playbook -i inventory/mycluster/hosts.ini cluster.yml
```

This command installs Kubernetes across the nodes, applies container runtime and Cloud Network Interface which in this case is calico, and gives us a ready to use cluster in which we can deploy our application. In order to verify the state of the nodes and pods, the following commands are used:

```
kubectl get nodes
kubectl get pods --all-namespaces
```

The status shown for both nodes and pods should be in the ready state; otherwise, we will not be able to proceed to the next steps.

4.3 Microservices Deployment

The application deployment in the Kubernetes environment follows a series of automated steps as defined by the project authors. First, the spring-petclinic namespace in Kubenetes is created. The associated services are then deployed, followed by the databases for each specific microservice using Helm, and finally the application itself is deployed via a custom script.

To set up the namespace for the Spring Petclinic application, the following command is executed.

```
kubectl apply -f k8s/init-namespace/
```

This creates the `spring-petclinic` namespace which isolates the application from other components.

Next, Kubernetes services used by the application are created with:

```
kubectl apply -f k8s/init-services
```

Verification of service deployment is performed by listing the services within the `spring-petclinic` namespace:

```
kubectl get svc -n spring-petclinic
```

The output is expected to list services such as `api-gateway`, `customers-service`, `vets-service`, `visits-service`.

Before deploying the databases, it is necessary to ensure that a single default StorageClass is active:

```
[breaklines=true]
kubectl get sc
```

The databases are then deployed using Helm via the Bitnami MySQL chart. The commands below deploy three separate database instances for the application:

```
helm repo add bitnami https://charts.bitnami.com/bitnami
helm repo update
helm install vets-db-mysql bitnami/mysql --namespace spring-petclinic
 ↪  --version 12.2.2 --set auth.database=service_instance_db
helm install visits-db-mysql bitnami/mysql --namespace
 ↪  spring-petclinic --version 12.2.2 --set
 ↪  auth.database=service_instance_db
helm install customers-db-mysql bitnami/mysql --namespace
 ↪  spring-petclinic --version 12.2.2 --set
 ↪  auth.database=service_instance_db
```

Finally, the application can be deployed using custom deployment manifests. The following command automatically runs the deployment process for all microservices.

```
./scripts/deployToKubernetes.sh
```

To verify that all application pods are running in the spring-petclinic namespace:

```
kubectl get pods -n spring-petclinic
```

This command should confirm that all pods such as api-gateway, customers-service, vets-service, visits-service, and the database pods are in a running state.

Lastly, to validate external access to the application, the external IP of the API Gateway service of NodePort type is obtained with:

```
kubectl get svc -n spring-petclinic api-gateway
```

The expected output indicates that the API Gateway has been assigned an external IP address, that enables external access.

4.4 Monitoring Stack Setup

In order to monitor the performance metrics of the microservices deployed on the Kubernetes cluster, kube-prometheus-stack Helm chart is utilized which is a bundle of Prometheus, Grafana, and the Prometheus Operator.

4.4.1 Installing kube-prometheus-stack

Before installation of the monitoring stack, the updated Prometheus Community Helm repository is added:

```
helm repo add prometheus-community
↪  https://prometheus-community.github.io/helm-charts
helm repo update
```

and then Prometheus stack is deployed in the monitoring namespace:

```
helm install my-prom-stack prometheus-community/kube-prometheus-stack
↪  --namespace monitoring --create-namespace
```

This command installs:

- **Prometheus Operator**: Manages CRDs such as ServiceMonitor and Prometheus.
- **Prometheus**: Responsible for scraping metrics from microservices and storing them.
- **Grafana**: Provides dashboards for data visualization.

4.4.2 ServiceMonitors for Microservices

In order for Prometheus to be able to scrape metrics from the microservices, objects named ServiceMonitor are defined and Prometheus looks for the matching labels, in our case `release: my-prom-stack`. Therefore, each ServiceMonitor includes:

```
metadata:
  labels:
    release: my-prom-stack
```

Additionally, the ServiceMonitor has to reference the same labels and ports that are used by each microservice's service. For example, if a microservice service is labeled `app:customers-service` and exposes metrics on port http at `/actuator/prometheus`, the corresponding ServiceMonitor configuration is provided in Appendix A.2.

Using this configuration helps Prometheus to scrape the microservice's runtime application metrics (e.g., memory usage, request rates, thread activity) on port `http` every 15 seconds.

Chapter 5

Experimental Results

5.1 Capacity Planning for Pods

Capacity planning for Kubernetes pods is essential to ensure the required Quality of Service (QoS). For a closed network of queues, as discussed in Section 3.3, in steady state, the joint distribution of number of requests at each node is of product form; therefore, the arrival process seen at node i can be treated as memoryless. According to this, in our architecture, each node's local behavior is the same as an $M/M/s$ queue. The objective is to determine, for every microservice, the minimum number of pods s such that the probability of queueing P_Q does not exceed a predefined threshold. Controlling P_Q ensures shorter expected queue times and, consequently, yields low response times. A practical rule of thumb is to keep P_Q less than 0.2 to maintain response times in an acceptable range. Table 5.1 is a summary of notations used in the following sections.

Notation	Description
R	Resource requirement ratio (λ/μ)
P_Q	Probability that an arriving task must queue
α	Maximum allowed queueing probability (QoS threshold)
c	Constant used in the square-root staffing rule
$\Phi(\cdot)$	Cumulative distribution function (CDF) of the standard Normal distribution
$\phi(\cdot)$	Probability density function (PDF) of the standard Normal distribution
T_{Q_i}	Mean queue time at node i

Table 5.1: Summary of notations

5.1.1 System Model and Assumptions

Let λ denote the average arrival rate to a microservice node, μ the processing rate of an individual pod, and s the number of pods provisioned for that node. The minimum number of pods required for stability is captured by the resource–requirement ratio

$$R = \frac{\lambda}{\mu} \tag{5.1.1}$$

with utilization $\rho = \lambda/(s\mu) < 1$ implying $s > R$.

5.1.2 Square–Root Staffing Rule

To satisfy the QoS constraint $P_Q \leq \alpha$, the square–root staffing rule [11] recommends

$$s = R + c\sqrt{R}, \tag{5.1.2}$$

where the constant c solves the nonlinear equation

$$\frac{c\,\Phi(c)}{\phi(c)} = \frac{1-\alpha}{\alpha}. \tag{5.1.3}$$

For a typical target of $\alpha = 0.20$, one obtains $c \approx 1.28$.

5.1.3 Arrival–Rate Estimation in a Closed Network

For a population of N users that alternate between think node with mean Z and services, the overall arrival rate is approximated by

$$\lambda = \frac{N}{Z + \sum_i 1/\mu_i + \sum_i T_{Q_i}}, \tag{5.1.4}$$

where the summation spans all microservice nodes. To estimate the maximum arrival rate (throughput), we set the queue waiting times to zero ($T_{Q_i} = 0$). The resource requirement for node i is therefore

$$R_i = \frac{\lambda}{\mu_i}. \tag{5.1.5}$$

5.1.4 Calculating Pod Requirements

Given R_i and the QoS threshold α, the number of pods for node i is

$$s_i = R_i + c\sqrt{R_i}. \tag{5.1.6}$$

Applying this rule to every microservice yields the minimal pod configuration that satisfies the global QoS goal while avoiding unnecessary over provisioning.

5.2 Experimental Results and Validation

This section presents comparative validation of the Stable Mean Value Analysis (SMVA), the discrete-event simulation model, and the performance observed in a real Kubernetes deployment of the Spring PetClinic application. The validation starts with determining service processing rates using load tests and operational laws, then proceeds to a direct comparison of key performance metrics such as throughput and mean response times across the three methods.

5.2.1 Estimation of Service Processing Rates

The service processing rates (μ values) for microservices customers-service, visits-service, and vets-service were estimated through individual load tests on the Kubernetes cluster. Each service was isolated and independently tested until it reached approximately 100% CPU utilization by increasing the load. At this saturation point, the operational law relating throughput (X_i), utilization (ρ_i), and service demand ($E[T_i]$) is:

$$\rho_i = X_i \cdot E[T_i] \tag{5.2.1}$$

For a fully utilized server ($\rho_i \approx 1$), the mean service time simplifies to:

$$E[T_i] = \frac{1}{X_i}, \quad \text{thus} \quad \mu_i = X_i. \tag{5.2.2}$$

Applying this, the following base processing rates were obtained (Table 5.2):

Service Name	**Processing Rate (μ, requests/sec)**
Customers-service	170
Visits-service	450
Vets-service	250

Table 5.2: Base processing rates obtained from individual load tests at saturation (100% CPU utilization).

5.3 Validation and Comparative Analysis

In this section, we validate the results obtained from the Square-Root Staffing capacity planning, the Stable Mean Value Analysis (SMVA), the Discrete-Event Simulation (DES), and the actual Kubernetes-deployed Spring PetClinic application. The validation includes a comparison of throughput and response times under varying workloads, using different pod configurations. Detailed screenshots from these validation tests are available in Appendix B.

5.3.1 Throughput Validation

Table 5.3 compares the throughput achieved by the SMVA algorithm and actual application under different user loads and pod configurations.

Test	Users (N)	Pods (Customers,Visits,Vets)	SMVA	PetClinic
1	50	1,1,1	80	80
2	50	2,1,1	81	81
3	100	1,1,1	152	130
4	100	2,2,2	162	150

Table 5.3: Throughput (requests/sec) comparison: SMVA and PetClinic App

5.3.2 Response Time Validation

Table 5.4 compares the response times obtained from the SMVA algorithm, Discrete Event Simulation (DES), and the actual PetClinic application under the corresponding test scenarios. All experimental runs used a mean think time (Z) of 0.6 seconds for the analytical and simulation models (SMVA, DES and capacity planning), while the actual PetClinic application used a lower think time of 0.3 seconds to accommodate network-induced latencies and closely match the analytical and simulation assumptions. Initially, using a mean think time of 0.3 in SMVA and DES resulted in significantly higher throughput values, almost twice those observed in the PetClinic load-test results. Therefore, to align the modeled throughput with the empirical results and to account for the network latencies, we increased the value of mean think time to 0.6 seconds.

Test	Users	Pods (Customer,Visits,Vets)	Service	SMVA	DES	PetClinic
1	50	1,1,1	Customers	10.8	12.9	11.0
			Visits	2.8	2.7	4.0
			Vets	5.8	5.9	7.0
2	50	2,1,1	Customers	6.7	7.2	7.2
			Visits	2.8	2.8	4.0
			Vets	5.8	5.7	7.0
3	100	1,1,1	Customers	41.0	42.0	43.0
			Visits	3.5	3.5	4.8
			Vets	9.9	9.2	10.2
4	100	2,2,2	Customers	8.4	9.8	7.2
			Visits	2.5	2.4	4.2
			Vets	4.9	4.6	6.2

Table 5.4: Response time comparison (ms) between SMVA, DES, and PetClinic

5.3.3 Validation of Capacity Planning Method

In the first round of tests, we ran SMVA, discrete-event simulation (DES), and the actual PetClinic application using a setup of one pod per node (Customer, Visit, and Vet services) with 50 users. The response times and throughput results from this initial test are recorded. In this simulation, each user iteratively issued 4 request. With a Then, we executed the capacity planning (CP) method with the same parameters (50 users, identical think time, and processing rates). The CP method suggested increasing the number of pods for Customer Service to two, while leaving other service pods unchanged. After applying these suggested adjustments and running the tests again, we observed a noticeable decrease in response times in the SMVA, DES, and PetClinic application.

The second round of tests involved scaling up the workload to 100 users while initially maintaining one pod per node. After recording the response times and throughput, we applied the CP method again, which recommended increasing the number of pods to two for each node. Implementing this new setup improved the response times in all three cases (SMVA, DES, and PetClinic).

The following table summarizes the total number of requests generated and simulation runtime for each test step.

Test Step	**Users**	**Total Requests**	**Simulation Runtime (min)**
1	50	20,000	3
2	50	20,000	5
3	100	40,000	2
4	100	40,000	5

Table 5.5: Simulation configuration and runtime per test step

Due to limitations in the underlying resources (virtual machines), we could not apply the recommendations of the capacity planning tool at larger scales. However, we validated these recommendations by applying them to the SMVA algorithm, and our observations confirmed that the total throughput matched the predictions from the capacity planning method, and response times improved when the suggested number of pods were applied. Table 5.6 summarizes the capacity planning method recommendations and the results obtained from the SMVA method when simulating a workload of 5000 users.

Metric	**Value**
Number of Users	5000
Recommended Pods (Capacity Planning)	
Customers-service	56
Visits-service	24
Vets-service	39
Throughput	
Predicted (Capacity Planning)	8167 rps
Obtained (SMVA)	8147 rps
SMVA Response Times	
Customers-service	6.500 ms
Visits-service	2.687 ms
Vets-service	4.518 ms

Table 5.6: Comparison of Capacity Planning Recommendations and SMVA Results at Larger Scale

As observed from the above results, the predicted throughput from the capacity planning method aligns closely with the prediction of the SMVA algorithm. Moreover, the response times remained relatively low and close to the mean processing times when we used the suggested pod configuration by the capacity planning code. For further validation, we reduced the pod instances for each of the services which led to noticeable increase in the overall response times. For example, when the number of pods was reduced to 48 for customers-service, 18 for vets-service, and 32 for visits-service, the corresponding response times shifted to 8.12 ms, 30.28 ms, and 7.67 ms, respectively. This experiment, further confirms the effectiveness and reliability of our proposed capacity planning approach.

Chapter 6

Conclusion

6.1 Main Conclusion

In this thesis, we have designed an approach to effectively plan the capacity of an SaaS-based application. The methods we have used consist of three main steps, two of which are complementary. We have first suggested utilizing a capacity planning approach, which implemented the square-root staffing rule for a network of queues, which in this case is a microservices application deployed on a Kubernetes cluster of nodes. The capacity planning suggests an optimal number of pods per service to maintain the performance within a reasonable range. This step ensures that we neither over-provision nor under-provision resources, thus managing costs effectively while adhering to Service Level Agreements (SLAs).

Afterwards, the suggested setup is tested in both analytical and simulation models to check the performance results, response time, and throughput. The analytical model is a closed Jackson network of load-dependent queues that is solved using the SMVA approach to estimate our performance metrics of interest. The SMVA algorithm was selected due to its computational simplicity, numerical stability, and suitability to handle load-dependent behavior in closed queueing networks.

The simulation model, on the other hand, is a discrete-event simulation of the application and the container orchestration engine that our apps are deployed and running on. Since analytical models may not be able to capture inherent complexities of applications and the influence of the environment and platforms on which these apps run, a simulation tool could be a convenient complementary way to account for these shortcomings. Our DES model is a straightforward tool for testing different routing strategies and also scenarios that involve a service that simultaneously interacts with multiple downstream services.

Furthermore, it has been an important matter for us to have validations of our models. We performed load tests on the ready-to-use Spring PetClinic Cloud project, which is a microservice-based web application. We deployed the app on a 2-node Kubernetes cluster and performed load tests to capture metrics such as HTTP requests, response times for each microservices pod, throughput, and CPU utilization of the pods. The results shown by these tests verify the validity and effectiveness of our capacity planning approach, as well as the queueing model and simulation-based performance analysis codes.

6.2 Future Work

Throughout this research, there have been points that could be noted to improve the effectiveness and generalization of our approach to a wider spectrum of applications and cover more granular behaviors of software systems.

Starting with the analytical method, we have used an approach that has inherent limitations regarding request flow. MVA-based approaches assume sequential request processing and are unable to handle parallel requests to downstream services since this violates the independence assumption required for queues. An important improvement here would be identifying approaches capable of handling fork-join scenarios, such as decomposition techniques.

Another matter for consideration involves careful characterization of workload distribution patterns. For simplicity, we have relied on Markovian assumptions like exponential processing times, which might not accurately reflect real system behaviors. To improve this, we could incorporate non-Markovian models, such as those with general distributions (e.g., phase-type distribution), or use empirical workload data to derive more accurate service-time distributions. This would increase the fidelity of our analytical and simulation models to real-world performance scenarios.

Moreover, another direction for the future work is to integrate the capacity planning approach with Kubernetes auto scaling feature. Kubernetes auto scaling, reactively adjusts the node and pod instances based on load variations and performance constraints. However, our method does this in a proactive way. Specifically, our capacity planning approach may be used as a baseline configuration for Kubernetes autoscalers. Furthermore, by dynamically monitoring the performance metrics of autoscaler with our analytical and simulation model results, we can refine the scaling policies.

A final aspect that we can improve is relevant to our capacity planning approach. Currently, our capacity planning tool only suggests an optimal combination of pods to keep application performance within a predefined threshold. However, we can enhance this by adding a scheduler that can decide where to optimally place the pods, based on underlying resource constraints. Having this scheduler alongside our capacity planning tool could result in better overall performance, reduced operational costs, and better resource utilization.

Appendix A

In this appendix, all code listings used in the thesis are provided.

A.1 Inventory Configuration

```
[all]

node1 ansible_host=10.0.0.1 ip=10.0.0.1

node2 ansible_host=10.0.0.2 ip=10.0.0.2

[kube_control_plane]

node1

[etcd]

node1

[kube_node]

node2
```

A.2 ServiceMonitor Configuration

The following configuration defines the ServiceMonitor for a microservice labeled `app:customers-service`. This configuration instructs Prometheus to scrape metrics from the microservice's endpoint `/actuator/prometheus` on port `http` every 15 seconds.

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: customers-service-servicemonitor
  namespace: monitoring
  labels:
    release: my-prom-stack
spec:
  selector:
    matchLabels:
      app: customers-service
  namespaceSelector:
    matchNames:
      - spring-petclinic
  endpoints:
    - port: http
      path: /actuator/prometheus
      interval: 15s
```

A.3 Source Code Repository

The source code and the simulation framework developed are available at this GitHub repository.

Appendix B

B.1 PetClinic Application Throughput

The figures demonstrate the throughput of PetClinic application microservices (Customers-service, Visits-service, and Vets-service) in a four-step load test. During each of the steps, the number of users or Pods changes, and the results for the changes in the throughput of each of the microservices are recorded.

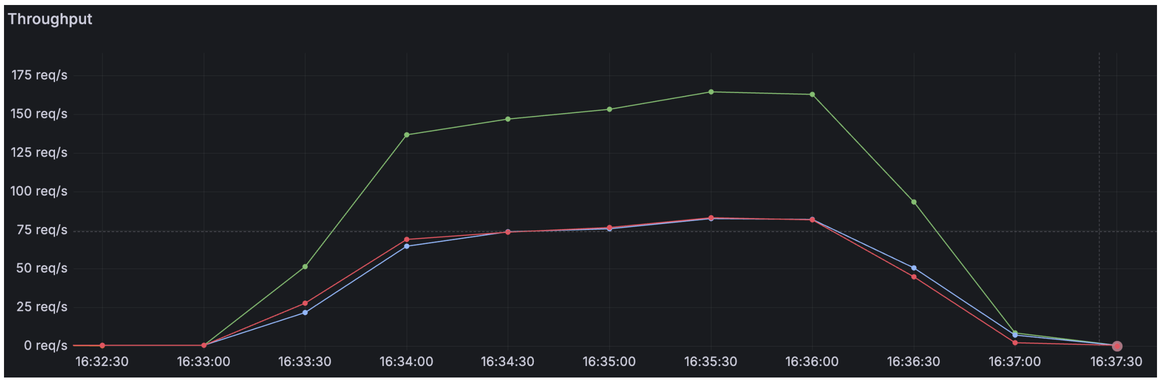

Figure B.1: PetClinic application throughput: Test 1 (50 users, 1 pod per service)

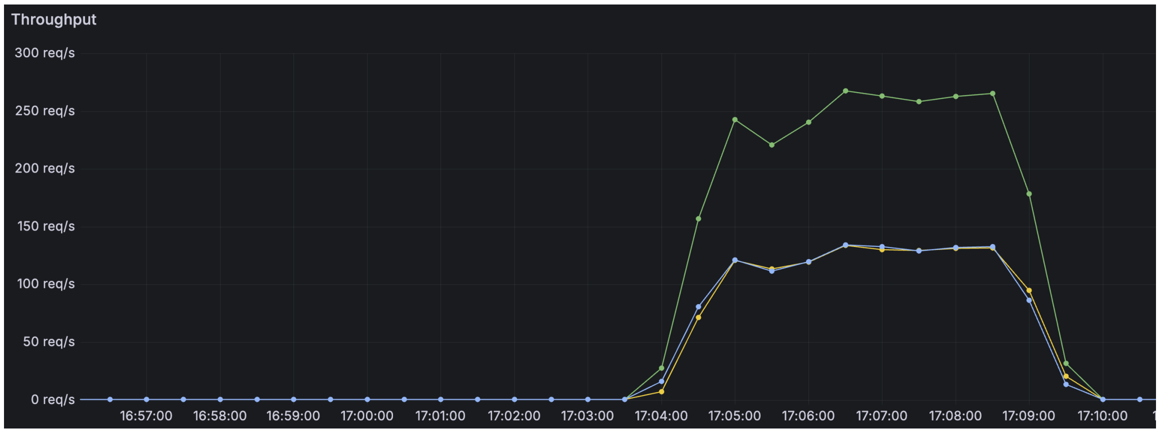

Figure B.2: PetClinic application throughput: Test 2 (50 users, 2 pods for customers, 1 for visits, 1 for vets)

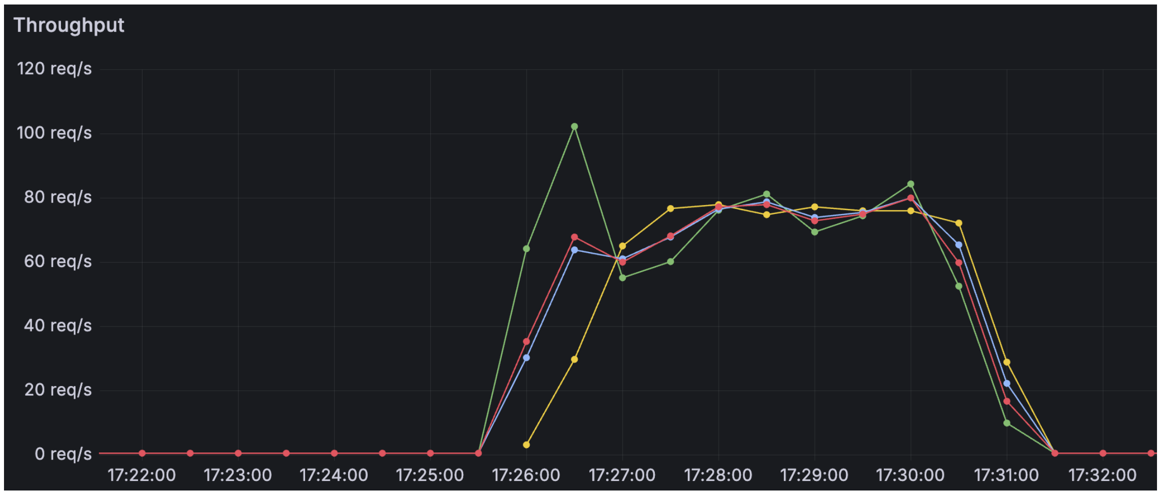

Figure B.3: PetClinic application throughput: Test 3 (100 users, 1 pod per service)

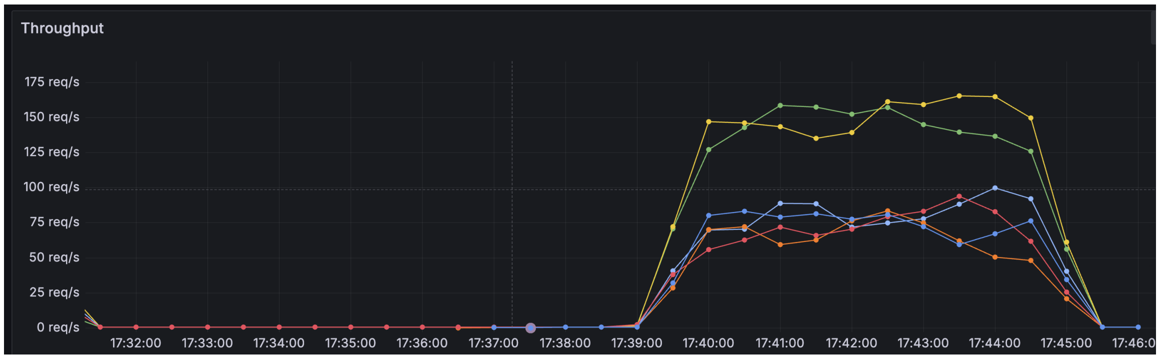

Figure B.4: PetClinic application throughput: Test 4 (100 users, 2 pods per service)

B.2 Discrete-Event Simulation Results (Response and Queue Times)

The following figures present the mean response times and queue times of the simulated PetClinic application in the DES framework, under the same previous setup.

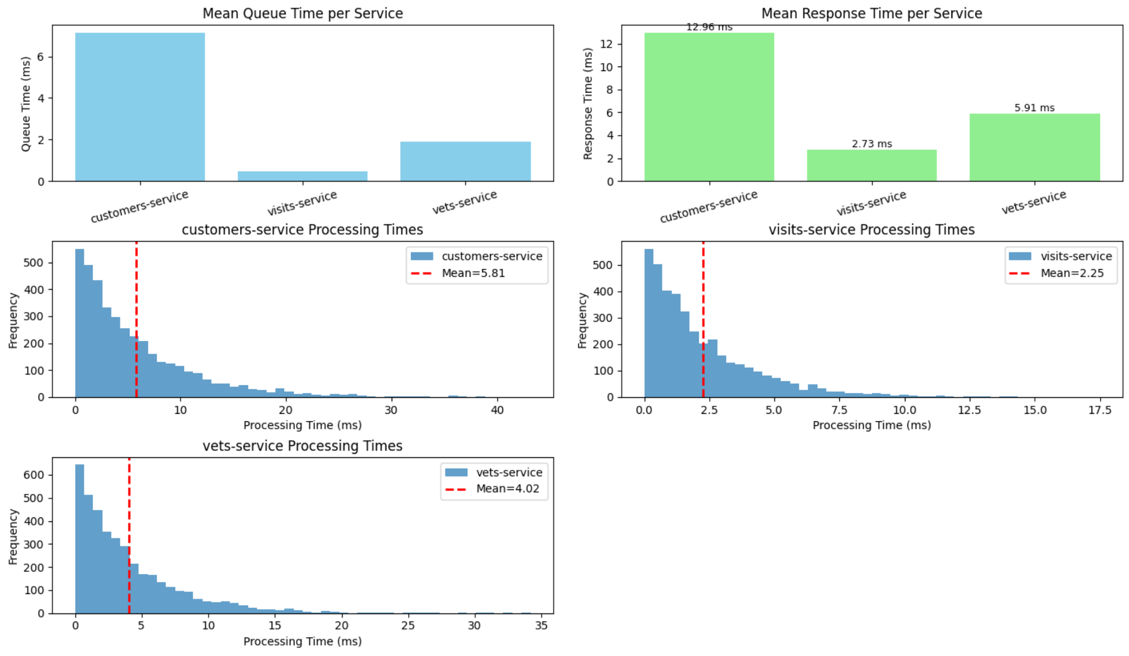

Figure B.5: Discrete-event simulation results: Test 1 (50 users, 1 pod per service)

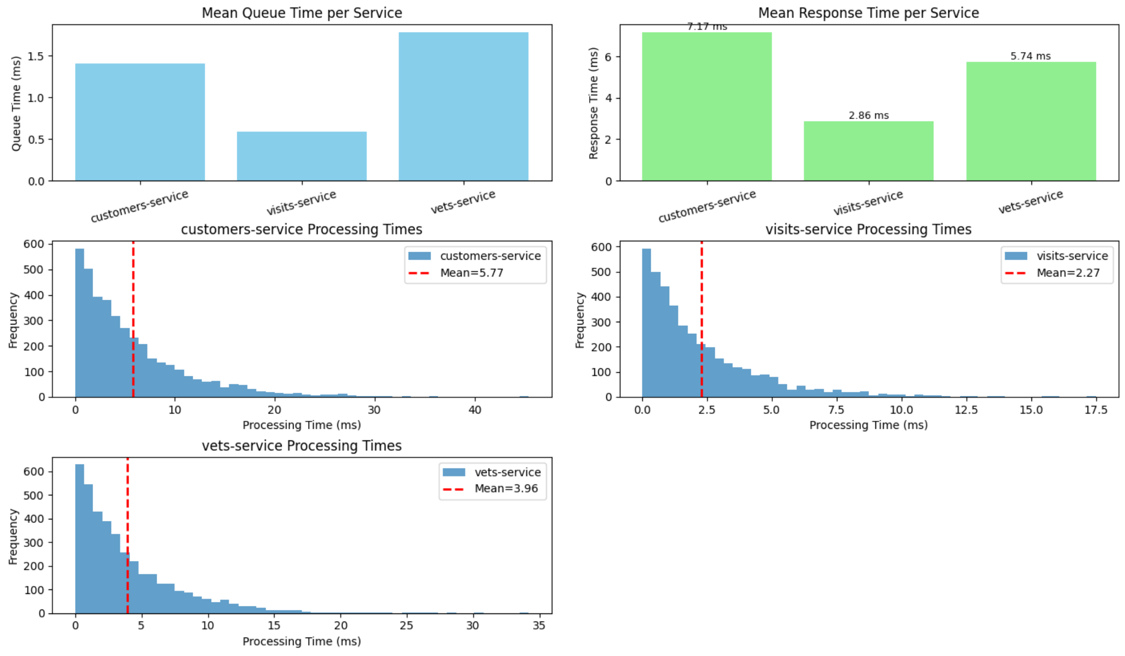

Figure B.6: Discrete-event simulation results: Test 2 (50 users, 2 pods for customers, 1 for visits, 1 for vets)

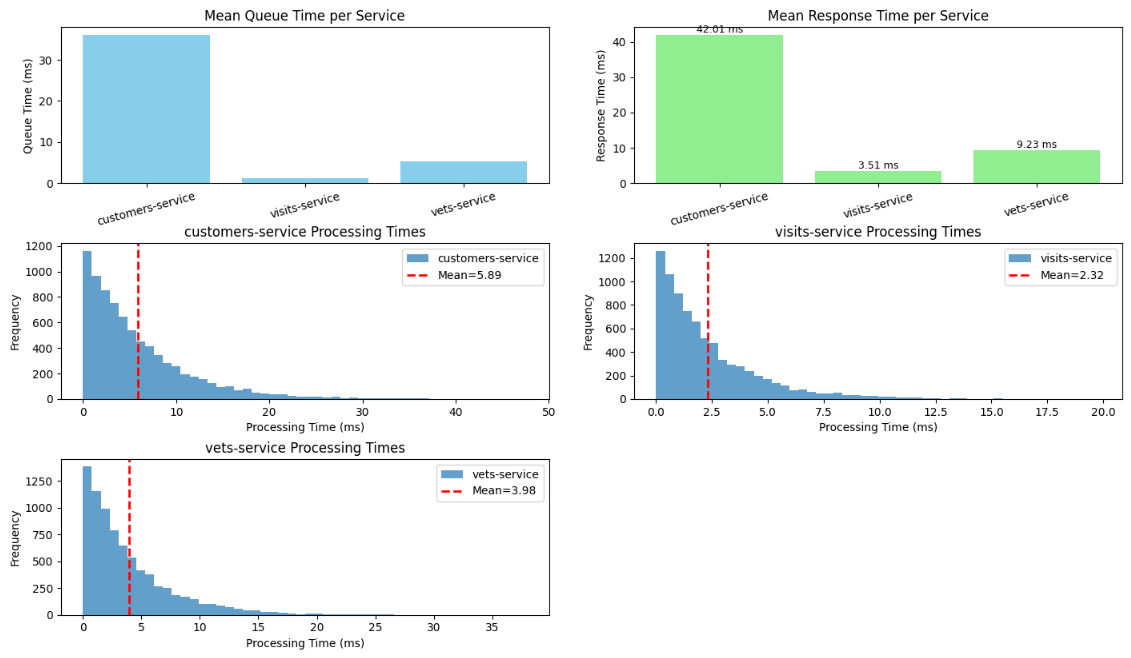

Figure B.7: Discrete-event simulation results: Test 3 (100 users, 1 pod per service)

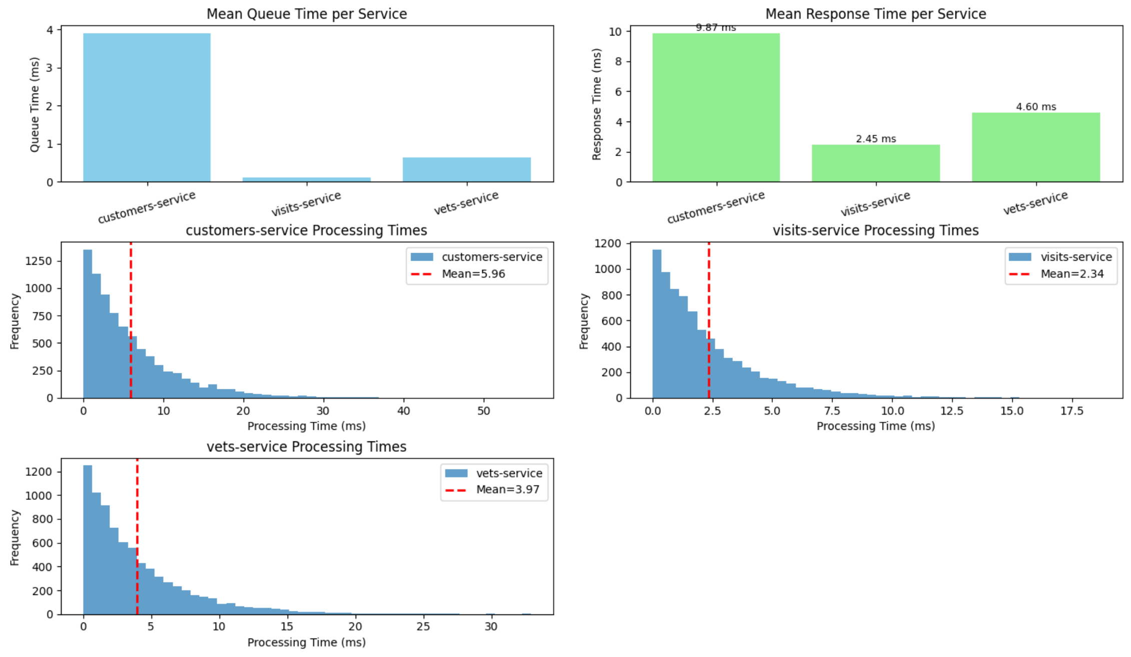

Figure B.8: Discrete-event simulation results: Test 4 (100 users, 2 pods per service)

Bibliography

[1] Y. Bard. Some extensions to multiclass queueing network analysis. In *Proceedings of the 3rd International Symposium on Modelling and Performance Evaluation of Computer Systems: Performance of Computer Systems*, pages 51–62, New York, 1979. North-Holland Publishing Co.

[2] M. Carvalho, D. A. Menascé, and F. Brasileiro. Capacity planning for iaas cloud providers offering multiple service classes. *Future Generation Computer Systems*, 77:97–111, 2017.

[3] G. Casale. On single-class load-dependent normalizing constant equations. *Performance Evaluation*, 65(11-12):844–858, 2008.

[4] A. C. H. Chen, M. C. H. Hsiang, and M.-Y. Wang. Efficiency analysis of microservices based on queueing models. In *2023 IEEE International Conference on Machine Learning and Applied Network Technologies (ICMLANT)*, pages 1–5, 2023.

[5] S. Chouliaras and S. Sotiriadis. Auto-scaling containerized cloud applications: A workload-driven approach. *Simulation Modelling Practice and Theory*, 121: 102654, 2022.

[6] J. Correia, F. Ribeiro, R. Filipe, F. Araújo, and J. Cardoso. Response time characterization of microservice-based systems. *IEEE Access*, 9:85232–85244, 2021.

[7] S. El Kafhali, I. El Mir, K. Salah, and M. Hanini. Dynamic scalability model for containerized cloud services. *Arabian Journal For Science and Engineering*, pages 10693–10708, 2020.

[8] E. Furman and A. Diamant. Optimal capacity planning for cloud service providers with periodic, time-varying demand. *Computers & Operations Research*, 129:105223, 2021.

[9] W. J. Gordon and G. F. Newell. Closed queueing systems with exponential servers. *Operations Research*, 15(2):254–265, 1967.

[10] J. Han, Y. Hong, and J. Kim. Refining microservices placement employing workload profiling over multiple kubernetes clusters. *IEEE Access*, 8:192543–192556, 2020.

[11] M. Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, 2013.

[12] J. R. Jackson. Networks of waiting lines. *Operations Research*, 5(4):518–521, 1957.

[13] J. R. Jackson. Jobshop-like queueing systems. *Management Science*, 10(1):131–142, 1963.

[14] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for*

Experimental Design, Measurement, Simulation, and Modeling. John Wiley & Sons, New York, NY, USA, 1991.

[15] A. Jindal, V. Podolskiy, and M. Gerndt. Performance modeling for cloud microservice applications. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering (ICPE '19)*, pages 25–32. ACM, 2019.

[16] H. Khazaei, C. Barna, N. Beigi-Mohammadi, and M. Litoiu. Efficiency analysis of provisioning microservices. In *IEEE 8th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 261–268. IEEE, 2016.

[17] Y. Kouki and T. Ledoux. Sla-driven capacity planning for cloud applications. In *IEEE 4th International Conference on Cloud Computing Technology and Science*, pages 135–140. IEEE, 2012.

[18] Kubernetes SIGs. Kubespray: Deploy a production ready kubernetes cluster. `https://github.com/kubernetes-sigs/kubespray`, 2024. Accessed: 2024-05-26.

[19] S. Marston, Z. Li, S. Bandyopadhyay, J. Zhang, and A. Ghalsasi. Cloud computing – the business perspective. *SSRN Electronic Journal*, 2010.

[20] D. A. Menascé, V. A. Almeida, and L. W. Dowdy. *Performance by Design: Computer Capacity Planning by Example.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[21] R. Pinciroli, A. Aleti, and C. Trubiani. Performance modeling and analysis of design patterns for microservice systems. In *2023 IEEE 20th International Conference on Software Architecture (ICSA)*, pages 35–46, 2023.

[22] P. Schweitzer. Approximate analysis of multiclass closed networks of queues. In *Proceedings of International Conference on Stochastic Control and Optimization*, pages 25–29, Amsterdam, 1979.

[23] A. A. Shahin. Enhancing elasticity of saas applications using queuing theory. *International Journal of Advanced Computer Science and Applications (IJACSA)*, 8(1):279–286, 2017.

[24] Spring PetClinic Project. Spring petclinic cloud. `https://github.com/spring-petclinic/spring-petclinic-cloud`, 2024. Accessed: 2025-06-18.

[25] A. Srivastava and N. Kumar. Queueing model based dynamic scalability for containerized cloud. *International Journal of Advanced Computer Science and Applications (IJACSA)*, 14(1):465, 2023.

[26] W. Tian, M. Xu, A. Chen, G. Li, X. Wang, and Y. Chen. Open-source simulators for cloud computing: Comparative study and challenging issues. *Simulation Modelling Practice and Theory*, 58:239–254, 2015.

[27] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. An analytical model for multi-tier internet services and its applications. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 291–302, 2005.

[28] Q. Zhang and D. G. Down. SMVA: A stable mean value analysis algorithm for closed systems with load-dependent queues. In *Systems Modeling: Methodologies and Tools*, pages 27–44. Springer International Publishing, 2019.

[29] Y. Zhang, Y. Gan, and C. Delimitrou. µqsim: Enabling accurate and scalable simulation for interactive microservices. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 212–222, 2019.