LUND UNIVERSITY

Automatic Differentiation over Fluid Models for Holistic Load Balancing

Heimerson, Albin; Ruuskanen, Johan; Eker, Johan

Published in:
2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)

DOI:
10.1109/ACSOSC56246.2022.00020

2022

Link to publication

Citation for published version (APA):
Heimerson, A., Ruuskanen, J., & Eker, J. (2022). Automatic Differentiation over Fluid Models for Holistic Load Balancing. In *2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)* (pp. 13-18) https://doi.org/10.1109/ACSOSC56246.2022.00020

Total number of authors:
3

General rights
Unless other specific re-use rights are stated the following general rights apply:
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Read more about Creative commons licenses: https://creativecommons.org/licenses/

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

Automatic Differentiation over Fluid Models for Holistic Load Balancing

Albin Heimerson, Johan Ruuskanen, Johan Eker

Abstract

Microservice applications consist of a set of smaller services interacting in a graph structure to deliver the full application. Jobs will traverse this graph in different paths, both depending on the type of job, but also depending on the current load of different service replicas. Different paths will incur different scenario-specific costs, dependent on, e.g., deployment and the underlying cloud system. In this paper, we demonstrate how automatic differentiation over data-driven fluid models can be used to optimize a running microservice application, by designing a load balancer that minimizes some holistic cost function under response time constraints. First, a fluid model describing the load in each service is learned through parsing tracing data from the application. We introduce a cost function based on performance metrics such as mean queue length and response time percentiles, all retrieved from the fluid model. By using automatic differentiation on this cost function, we can find the gradient of the cost with respect to the load balancing parameters. This enables us to update these parameters, using e.g. gradient descent, in a manner that steers the application towards a more optimal setting. In an experimental evaluation on a small microservice application running on Ericsson Research Datacenter, it is shown that the method can quickly step towards optimal values while supporting complicated cost functions such as solutions to a system of ordinary differential equations.

1 Introduction

Cloud computing started with server consolidation scenarios where servers could be seamlessly migrated to virtual machines hosted in public datacenters. Since then, it has grown into a large ecosystem with a wide range of services such as managed databases, IoT platforms, machine learning frameworks, etc.

Recently, container technologies have gained popularity as a more lightweight alternative to virtual machines. So-called microservices [1] has become the dominant cloud service architecture. A microservice is typically a smaller service with a single purpose. A full-fledged cloud service is then constructed from a set of microservices. Using a microservice architecture has the benefits of scaling, migrating or replicating parts of the functionality on an individual basis, allowing for very flexible orchestration strategies. Kubernetes[1] is a portable, extensible, open-source platform for managing containerized workloads and microservices.

Essentially, cloud computing is a business model where the focus is on compute density, resource pooling, and elasticity, to maximize utilization and minimizing cost. Elasticity is a key feature of cloud computing, as it allows cloud services to scale by dynamically adding or removing resources to maintain stable performance levels despite fluctuations in the workload. These resources can be rapidly provisioned and released with minimal management effort or service provider intervention.

Commonly, cloud services are associated with a Service Level Agreement (SLA) consisting of multiple Service Level Objectives (SLOs) stipulating expectations on availability, performance, latency, etc. The service provider has to balance the cost of allocated resources against the cost of violating the SLOs. An ideally configured service is meeting the SLOs with a minimum amount of allocated resources.

With the increased demand for low and predictable latency, due to, for example, new use cases such as cloud based automation, IoT, XR (eXtended Reality), and VR (Virtual Reality), edge computing has gained traction as a means of bringing compute resources closer to the user. A combination of centralized cloud and edge compute nodes offers a distributed compute platform which allows for trade-offs between latency, cost, etc., and makes it possible to configure a cloud service to deliver solutions at the right service level at the right cost.

In this work we consider microservices that are distributed across multiple *sites* such as edge nodes, local

[1] https://kubernetes.io

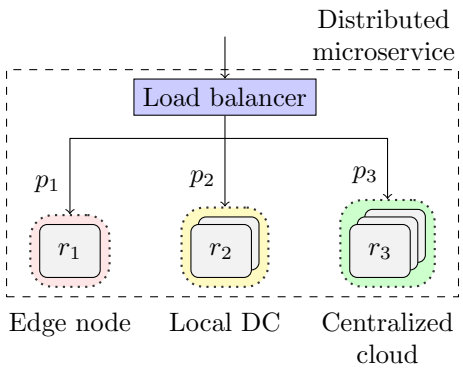


Figure 1: A distributed microservice with replica sets r_i over multiple different sites. The load balancer distributes the incoming load according to a weighted random scheme with probabilities p_i over the available replica sets. This can then be part of a larger application of multiple communicating microservices.

datacenters, and centralized clouds. Each of these sites comes with different cost and performance. The challenge we address is how to distribute the incoming workloads over the different sites and satisfy the SLOs while minimizing cost.

1.1 Contributions

The load balancer in Fig. 1 distributes the incoming workload over replica sets, replicated instances of a service, located in the different sites. In the proposed model, the traffic is directed to a given site with probability p_i. Due to differences in communication delays, available capacity etc., the access latency and the cost vary among each site. The goal is to select the probabilities p_i to minimize the total cost while also keeping the response time below a threshold. The incoming workload is constantly changing, meaning that any strategy must be able to adapt to variations in traffic. In this paper, we use a fluid model of the environment and perform online optimization of a cloud application using automatic differentiation. We specifically focus on adapting the load balancer probabilities in order to minimize some cost function, but the technique itself is more general.

The contributions of the paper are the following:

- A method for optimal weighting, with respect to some cost function, of the random load balancing policy is provided. The minimization is performed online using gradient stepping via automatic differentiation of said cost function.
- We propose a cost function that is based on the solution of a fluid model, and can thus take into account transient and stationary queue lengths, and constraints on e.g. response time percentiles.
- We demonstrate the solution in a real-world setting with an application distributed over three different Kubernetes clusters connected over a network gateway to simulate delays between our "sites". The setup is running in the Ericsson Research Datacenter.

2 Background and related work

2.1 Microservice fluid model

Queuing theory is a popular method for modeling cloud systems, especially considering the queuing disciplines *first-come first-served* (FCFS), *processor sharing* (PS) and *delay* (INF) [2, 3]. Dependencies between resources and servers can be modeled by joining multiple queues, forming a *queuing network*. Unfortunately, queuing networks seldom have closed-form solutions, and evaluation by simulation is often prohibitively computationally expensive for real-time usage. Instead, important performance metrics are often approximated using different methods. Most popular among these for stationary solutions of product-form networks is the mean-value analysis (MVA) and its extensions [4]. Another common method is the fluid model, which approximates the time evolution of the mean queue length as a set of ordinary differential equations (ODE). Thus, fluid models can yield transient solutions, that do not require product-form assumptions and are fast to solve using modern methods.

In this paper, we adopt the fluid model for microservice applications introduced in [5], based on the mean-field approximation of a simplistic queuing network representation of the application. In short, the queuing network models each replica to every service as a multiclass PS queue and each replica-to-replica delay as a multiclass INF queue. Each processing stage of a request across the entire microservice application is then modeled as a path over the classes in the network. An illustration of this can be seen in Fig. 2. Poisson arrivals are assumed, and the service time in each class is allowed to follow a *phase-type* distribution [6], representing the service time as the time-to-absorption in an internal Markov chain. The transient states of this chain are referred to as *phase states*. Although the individual replica models are simple, this fluid

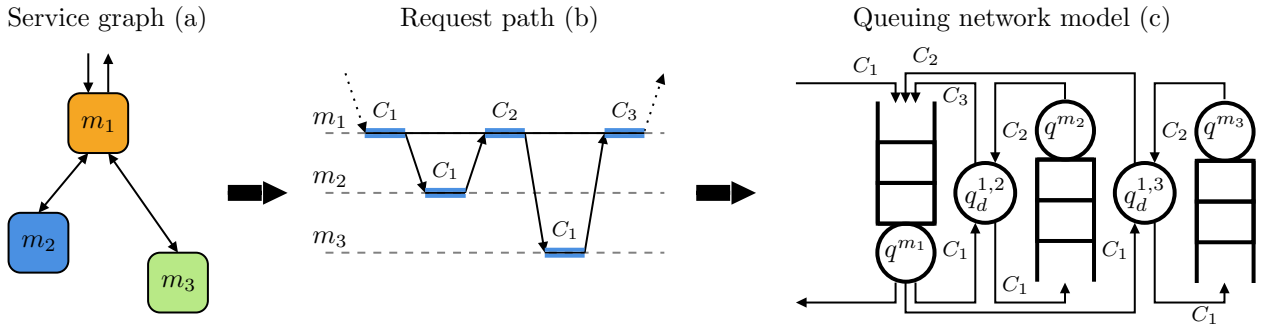


Figure 2: The transformation from a microservice graph (a) to the corresponding queuing network model (c). The example service graph consists of 3 services, each with 1 replica, where external requests arrive to the service m_1. Requests in m_1 performs two remote calls to m_2 and m_3. In (b), the path that all external requests must traverse is shown together with the corresponding class decomposition in a spanning diagram. In (c), the replicas have been replaced by the multiclass PS queues $q^{m_1}, q^{m_2}, q^{m_3}$, the replica-to-replica connection delays by the INF queues $q_d^{1,2}, q_d^{1,3}$. Each arrow displays a class-to-class transition and is marked with the downstream class it connects to.

model can capture quite general graphs of microservices, and be completely extracted at runtime from commonly collected[2] tracing data. However, due to its simplicity, it can experience modeling errors when predicting too far from the operating point where it was fitted.

Before introducing the fluid model, some definitions are needed. Let $\mathcal{Q}$ be the set of queues, $\mathcal{C}$ the set of classes and $\mathcal{S}$ the set of phase states in the network. Each queue q is assumed to have a unique set of classes $\mathcal{C}_q$, and each queue/class pair $(q, c) \in (\mathcal{Q}, \mathcal{C}_q)$ is assumed to have a unique set of phase states $\mathcal{S}_{q,c}$. Further, let k_q be the number of processors in queue q, $\boldsymbol{\lambda} \in \mathbb{R}_+^{|\mathcal{C}|\times 1}$ the Poisson arrival rates to each class and $\boldsymbol{P} \in \mathbb{R}^{|\mathcal{C}|\times|\mathcal{C}|}$ the class-to-class routing probability matrix. Finally, let $\boldsymbol{\Psi} \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{S}|}, \boldsymbol{B} \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{C}|}, \boldsymbol{A} \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{C}|}$ be the parameter matrices for the phase type distributions of each class, stacked into block diagonals in an appropriate order.

The *smoothed mean-field fluid model* [7] can then be introduced. It approximates the mean request population in each phase state $\mathbb{E}[\boldsymbol{X}(t)] \in \mathbb{R}^{|\mathcal{S}|\times 1}$ with the solution $\boldsymbol{x}(t) \in \mathbb{R}^{|\mathcal{S}|\times 1}$ to the following system of ODEs.

$$\frac{d\boldsymbol{x}}{dt} = \boldsymbol{W}^T \theta(\boldsymbol{x}, \boldsymbol{z}) + \boldsymbol{A}\boldsymbol{\lambda} \tag{1}$$

where $\boldsymbol{x}(0) = \boldsymbol{X}(0)$, $\boldsymbol{W} = \boldsymbol{\Psi} + \boldsymbol{B}\boldsymbol{P}\boldsymbol{A}^T$, and the function $\theta_i(\boldsymbol{x}, \boldsymbol{z}) = x_i g_{Q(i)}(\boldsymbol{x}, \boldsymbol{z})\ \forall i \in \mathcal{S}$ where

$$g_q(\boldsymbol{x}, \boldsymbol{z}) = \frac{1}{\left(1 + \left(k_q^{-1} \sum \boldsymbol{x}_q\right)^{z_q}\right)^{1/z_q}} \qquad q \in \mathcal{Q} \tag{2}$$

Here, $Q(i)$ is a function that maps a phase state to its parent queue and z_q the smoothing parameters in the queue q. This parameter aims to improve accuracy in mean-field fluid models, and can quickly be fitted to data using bisection search.

2.2 Automatic differentiation

Automatic differentiation (AD) is a technique to evaluate the derivatives of functions defined by computer programs. The basic idea is to apply the chain rule to the code in order to reduce it to simpler expressions where the derivative of each individual operation is easily defined. To automate this process there are a few different methods, but for this work we choose an implementation using a type of *dual numbers* [8].

Dual numbers are a convenient way to propagate information about the value of an expression, as well as the derivative of the expression, at the same time. So if we have a program where the higher level code is agnostic to type, and the lower level operations are defined for dual numbers, both the value and the derivative can be calculated in one go by supplying parameters as dual numbers.

Automatic differentiation distinguishes itself from numerical differentiation by being *exact* in the mathematical sense. While numerical differentiation approximates derivatives using finite differences, and will thus suffer

[2] `https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/observability/tracing#what-data-each-trace-contains`

from the inaccuracies coming from those techniques, automatic differentiation will do the calculation for the exact mathematical expression. Numerical differentiation also has the disadvantage that higher order derivatives are complex and involve larger errors, while gradients over multiple variables can become inefficient as the number of variables grow, since the number of evaluations of the function then also grows [9].

2.3 Load balancing

In this paper, we consider the load balancing problem of assigning a stream of requests to a set of replicas of a service. There is no magic bullet, as the "best" choice of load balancing strategy depends on characteristics of the workload and system, as well as the desired outcome [10]. Arguably, the most commonly used strategies are *round-robin* (RR), *random* and *join-shortest-queue* (JSQ) (also known as Least-Connected/Least-Request) and their weighted counterparts [11, 12], available in many modern software tools such as the proxies Envoy[3] and Nginx[4]. JSQ has in general better performance than the other two, and is for PS queues near optimal in minimizing the total mean response time [13]. However, it requires knowledge about the current state in each replica at each decision and is thus cumbersome to implement at scale. To handle this state knowledge drawback, many improvements have been suggested such as *power-of-d* (SQ(d)) [14, 15], *join-idle-queue* (JIQ) [16] and *join-the-best-queue* (JBQ) [17].

Although JSQ and its extensions promises a good mean response time in a load balancing scenario, they are difficult to analyze under more general settings and costs. We thus restrict ourselves to study the weighted random strategy, which allows us to express the application as a queuing network with probabilistic routing and subsequently the optimization problem as finding the probabilities that minimizes our cost. This problem has been extensively studied in the queuing theory community, but in order to make it feasible only simple cost functions and performance metric constraints (if any) are typically considered for specific types of queues and networks. We will here state some notable results, and refer to the references within for a more encompassing description of this field. Regarding open networks, response time minimization has been studied in e.g. [18] considering M/M/1 FCFS queues with link constraints using flow deviation, [19] considering M/G/1 FCFS queues over a weighted sum of response times, and [20] considering G/G/1 queues of either FCFS and PS discipline with constraints on response time variance. For closed networks, throughput maximization has been studied for product-form networks in e.g. [21] using gradient stepping as the gradient is readily obtained from the MVA algorithm, and [22] using closed-form heuristics based on heavy-traffic limits which were later explored in [23] for heuristic weighting of the RR strategy. In [24] it is shown that for product-form networks with general cost function and constraints on queue states, a Nash equilibrium can be obtained via deterministic routing. Further, in [25, 26] *model-predictive control* (MPC) over the mean-field fluid model for a closed network was studied in order to minimize response time by tuning both routing probabilities and autoscaling.

Compared to these results, our approach of gradient stepping using automatic differentiation is more general. It allows for arbitrarily defined costs and constraints from any performance metric that is obtainable from our fluid model. Further, the fluid model allows us to consider both transient and stationary metrics, and non-product-form networks of INF and PS queues. However, in adopting such a general approach, we forgo any theoretical results on optimality bounds, feasibility, and convergence speed.

3 Application model

We consider a cloud application subjected to requests from external users with an exponential inter-arrival time, i.e. Poisson arrivals, and where there are (soft) constraints on certain performance metrics, e.g. response time percentiles, via SLOs. The application is assumed to consist of multiple distributed microservices interacting in a graph structure. Each microservice is further assumed to have a set of *replicas*. The replicas of each service are allowed to span multiple placement possibilities, e.g. machines, clusters or even different sites, each associated with their own communication delay. The service of requests incurs a certain *cost* for the application owner, depending on things such as the cost of electricity, availability, the specific cloud provider and more. This cost will be highly specific to the application and deployment. What we will assume is that the cost is tied to *where* the requests are executed among the placement possibilities.

The task at hand is to minimize the total cost of run-

[3] `https://www.envoyproxy.io/docs/envoy/v1.5.0/intro/arch_overview/load_balancing`

[4] `https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/`

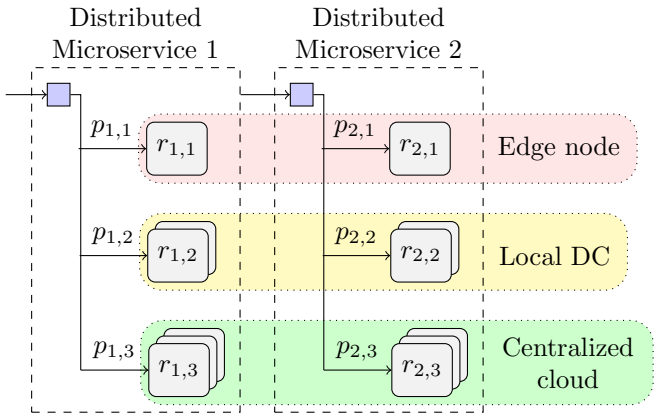


Figure 3: An application consisting of two distributed microservices, each with an internal load balancer and replicas $r_{i,j}$ spread over different sites. The control variables are the set of all routing probabilities for the load balancers, where $\boldsymbol{p}_i = [p_{i,1}, p_{i,2}, p_{i,3}]$ is the distribution for microservice i.

ning the application, while not violating the constraints, by tuning parameters related to the application deployment and management. In order to effectively determine such parameters, a model can be used to estimate the impact of them on both cost and constraints. Moreover, given such a model, it is possible to use automatic differentiation to differentiate the cost derived through this model w.r.t. these parameters. Potentially, one could then devise a control strategy to steer the entire application towards an operating region of less cost while keeping clear of the constraint limits. In this paper, we exemplify such a procedure by considering load balancing between the different replica sets in the application. Thus, our parameters to tune, or *control variables*, will be the set of all load balancing parameters.

An illustration of this kind of environment is presented in Fig. 3. The load balancer is placed outside any site for conceptual understanding, but for a practical implementation, the load balancer would exist on all sites of the microservice to reduce unnecessary network traffic. This would also allow a more general approach where each individual replica set has its own parameters for load balancing, which could further reduce redundant traffic between sites.

As our model, we adopt the microservice fluid model introduced in Section 2.1. In this model, routing between the services is fully determined by the class-to-class transitions described in the routing probability matrix $\boldsymbol{P}$. Hence, it captures the more general model where all replicas use a random policy for load balancing requests, and the probabilities are given by the rows of $\boldsymbol{P}$. As the model captures different request types as stand-alone classes, multiple non-zero values in the rows of $\boldsymbol{P}$ only occurs for load balancing purposes and are thus easy to find. The set of all non-zero routing probabilities is captured in the following definition.

Definition 1. *Let $\boldsymbol{p}_i$ be a vector of the non-zero probabilities in the i'th row of $\boldsymbol{P}$, and let $\mathcal{P} = \{\boldsymbol{p}_1, \boldsymbol{p}_2, \dots\}$ be the set of all probability vectors in the system.*

The elements of $\mathcal{P}$ are thus our control variables. Due to probability constraints, the following can be stated.

Remark 1.1. *As $\sum \boldsymbol{p} = 1 \ \forall \boldsymbol{p} \in \mathcal{P}$, connections with no load balancing, i.e. where $\boldsymbol{p}$ only has a single downstream, are fixed.*

Changing the elements of $\mathcal{P}$ will affect the solution of the fluid model (1), and we denote this dependence as $\boldsymbol{x}(t \mid \mathcal{P})$.

3.1 Obtaining desired metrics

The cost and constraints for the application are based on different performance metrics. These metrics needs to be derived from our model in order to create a differentiable mapping from control variables to costs and constraints. We base the cost on the mean number of requests present in each replica, and have a single constraint on a response time percentile.

3.1.1 Mean requests in replicas

As each replica is modeled by a single queue, the mean request present at time t can be approximated from the fluid model (1) by summing over all the phase states present in that queue, i.e.

$$x_q(t \mid \mathcal{P}) = \sum_{i \in \mathcal{S}_q} x_i(t \mid \mathcal{P}) \qquad q \in \mathcal{Q} \tag{3}$$

Further, let $\boldsymbol{x}_Q \in \mathbb{R}_+^{|\mathcal{Q}| \times 1}$ be the vector of all modeled mean requests populations in each queue q in the network.

3.1.2 Response time percentiles

As shown in [7], it is possible to obtain an approximation of the response time CDF given our fluid model. Let $\boldsymbol{\pi}(t) \in \mathbb{R}_+^{|\mathcal{S}| \times 1}$ be the probability vector of finding a request in the corresponding phase state after t time units,

and $\beta \in \mathbb{R}_+^{|\mathcal{C}|\times 1}$ the probability vector of the request entering the corresponding class at $t = 0$. The probability of remaining in the queuing network at time t can then be approximated with the following ODE,

$$\frac{d\boldsymbol{\pi}}{dt} = \boldsymbol{W}^T D^{g(\boldsymbol{z})} \boldsymbol{\pi}(t), \qquad \boldsymbol{\pi}(0) = \boldsymbol{A}\beta \tag{4}$$

where $D^{g(\boldsymbol{z})} \in \mathbb{R}_+^{|\mathcal{S}|\times|\mathcal{S}|}$ is a diagonal matrix with elements $D_{ii}^{g(\boldsymbol{z})} = g_{Q(i)}(\boldsymbol{z})$, i.e. (2) evaluated at the stationary solution $\boldsymbol{x}^*$ given some $\boldsymbol{z}$. As (4) is a linear system, it has a closed form solution. An approximation of the percentile φ_α can then be obtained by either bisection search over this closed-form solution, or by evaluating (4) and finding the t such that $\sum \boldsymbol{\pi}(t = \varphi_\alpha) = 1 - \alpha$. As the percentile and its approximation depends on $\boldsymbol{x}^*$, they also depend on the choice of $\mathcal{P}$ which we denote as $\varphi_\alpha(\mathcal{P})$.

4 Holistic load-balancing

The fluid model allows us to pose an idealized *optimal control* version of the original cost minimization problem, assuming a set of load balancing probability trajectories $\mathcal{P}(t)$ and some cost function $L_o(\cdot)$, as follows

$$\begin{aligned} \min_{\mathcal{P}(t)} \quad & \int L_o\left[t, \boldsymbol{x}_Q(t), \mathcal{P}(t)\right] dt \\ \text{subject to } & d\boldsymbol{x}/dt = (1) \\ & \sum \boldsymbol{p}(t) = 1 \ \forall t, \boldsymbol{p} \in \mathcal{P} \\ & \varphi_\alpha\left[\mathcal{P}(t)\right] \leq \varphi_{\lim} \ \forall t \end{aligned} \tag{5}$$

By minimizing over all $\mathcal{P}(t)$ we try to directly find the optimal load distribution that over time minimizes selected cost. Although possible, the model will become less accurate the further $\mathcal{P}(t)$ is from what it was when recording the data used to fit the model, and thus the predicted optimal is also less accurate. Further, as cloud systems are inherently dynamic due to resource contentions, workload changes, migrations, or even malfunctions, the optimal $\mathcal{P}(t)$ will generally not be convergent as the system is subjected to both slow and abrupt changes over time. This necessitates an optimization scheme where we perform simultaneous online optimization and model tuning in order to both update $\mathcal{P}(t)$ in a robust manner and adapt to changes in the system. To create such an algorithm, some adjustments to (5) are first needed.

4.0.1 Iterative model refitting & optimization

The model will become less accurate as the operating state change from what was used to train the model. Thus, we cannot be certain that a control action based on $\boldsymbol{x}(t)$, in regions beyond the current state, will have the expected effect on the real system. Taking such an action increases the risk of accidental violation of constraints or unstable control.

This can be remedied by continuously updating the model. But due to the fast timescale of the system dynamics compared to the time needed to gather enough data for an accurate model fitting using the scheme from [5], robust online model tracking at the necessary speed becomes a non-trivial problem. Instead, one possible simple solution is to update $\mathcal{P}$ in discrete steps, where between each step the system is monitored in order to gather enough data to refit the model before deciding the next $\mathcal{P}_k$. By bounding how far $\mathcal{P}_k$ can move from $\mathcal{P}_{k-1}$, it is then possible to assure that the system moves within some vicinity of the current operating condition where the model is accurate, increasing robustness against accidentally violating the constraints. We denote step k as the period between t_{k-1} and t_k, the sample time as $h = t_k - t_{k-1}$, the data gathered during step k as $\mathcal{D}_k$, and the model re-fitted on $\mathcal{D}_k$ as $\dot{x} = F_k(\boldsymbol{x})$, where $F_k(\boldsymbol{x}) = \boldsymbol{W}^T\theta(\boldsymbol{x}, \boldsymbol{z}) + \boldsymbol{A}\boldsymbol{\lambda}$ according to (1).

This iterative scheme will however result in a slower control action, and thus a potentially slower convergence of the cost function minimization, than fully relying on the model to decide some trajectory $\mathcal{P}(t)$ in a single step. In fact, the system will reach a stationary operating condition before deciding the next $\mathcal{P}_k$ as this is required to reliably re-fit the model. But as the overall goal is to minimize cost of a running system over a potentially very long time horizon, this slowdown is acceptable as long as the system is not subjected to too many disturbances of too high frequency.

4.0.2 Optimizing over weights

Optimizing directly over the probabilities becomes cumbersome, as we need to adhere to the constraint $\sum \boldsymbol{p} = 1$ $\forall \boldsymbol{p} \in \mathcal{P}_k$. Instead, it is possible to optimize over weight vectors $\boldsymbol{w}$ and enforce the constraint via the *softmax* function $S(\boldsymbol{w})$ [27], where

$$S_i(\boldsymbol{w}) = \frac{\exp(w_i)}{\sum_j \exp(w_j)} \tag{6}$$

The softmax function maps vectors defined on $\mathbb{R}^n$, to vectors whose elements are allowed to take values in the interval $[0, 1]$ and that fulfills $\sum_i S_i(\boldsymbol{w}) = 1$. Further, the softmax function preserves the order of the vector element quantities, i.e. if $w_i \geq w_j$, then $S_i(\boldsymbol{w}) \geq S_j(\boldsymbol{w})$. We let each $\boldsymbol{p} \in \mathcal{P}_k$ be associated with a weight vector $\boldsymbol{w}$, and introduce the following.

Definition 2. *Let $\mathcal{W}_k = \{\boldsymbol{w}_1, \boldsymbol{w}_2, \dots\}$ be the set of all weight vectors at time step $k+1$, and let $\mathcal{P}(\mathcal{W}_k) := \{S(\boldsymbol{w}) \; \forall \boldsymbol{w} \in \mathcal{W}_k\}$.*

To clarify the subscript, the set $\mathcal{W}_k$ will be decided based on data $\mathcal{D}_k$ that is gathered during step k, i.e. between $\{t_{k-1}, t_k\}$, using $\mathcal{W}_{k-1}$. It will then be used during step $k+1$.

4.0.3 Limiting the step size

A natural way of managing the step sizes would be to introduce some cost on the difference between $\mathcal{W}_k$ and $\mathcal{W}_{k-1}$. However, certain disturbances such as an increase in the load would increase the overall cost of the system, and thus change the relative step sizes if care is not taken. Instead, we will manage the step size limits by introducing the following constraint on the 2-norm on the change in probability over all weight vectors

$$\sqrt{\sum_{\boldsymbol{w} \in \mathcal{W}_{k-1}} ||S(\boldsymbol{w}) - S(\boldsymbol{w}^+)||_2^2} \leq d\mathcal{W}_{\lim} \tag{7}$$

where $\boldsymbol{w}^+$ is the corresponding updated $\boldsymbol{w}$ in $\mathcal{W}_k$.

4.0.4 Reworking the percentile constraint

Due to the dynamic nature of cloud systems, we cannot guarantee that any constraints based on performance metrics can actually be fulfilled at any time step. A disturbance might arise that pushes the system to an operating region where a constraint is violated. This can also happen if we are unlucky with the robust stepping of $\mathcal{P}_k$, although we will still be in the vicinity of the constraint limit.

The optimization algorithm must thus be able to handle such cases, and quickly drive the system back to a viable operating region. To do this, we can remove the constraint and instead heavily penalize the cost function in the case of violation by e.g. an additive cost function term $L_\varphi[\mathcal{P}(\mathcal{W}_k)]$ using a *penalty function*. The important thing is that the penalization should be negligent as long as the constraint is not violated, sharply increases around the constraint limit, and continue to quickly grow the further from the limit the system moves. This will ensure that the gradient of the cost function points the parameters towards viable operating regions.

4.0.5 New cost function

At time step k, the refitted model $F_k(\boldsymbol{x})$ can then be used to determine the next $\mathcal{W}_k$ using the following new cost function

$$L_k(\mathcal{W}) = \int_0^{t_f} L_q[t, \boldsymbol{x}_Q(t_k + t), \mathcal{P}(\mathcal{W})]\, dt + L_\varphi[\mathcal{P}(\mathcal{W})] \tag{8}$$

subjected to $\dot{\boldsymbol{x}} = F_k(\boldsymbol{x})$ where $\boldsymbol{x}(t_k)$ is set to the stationary solution of $\boldsymbol{x}[t \mid \mathcal{P}(\mathcal{W}_{k-1})]$, and the step size constraint (7). As we obtain transient values from the fluid model, we can minimize over these given some arbitrary cost function $L_q(\cdot)$ from current time t_k over some time horizon t_f. In general, as the system will reach a stationary state before the next action is taken, we should let t_f be large enough to capture the stationary values of $\boldsymbol{x}_Q(t)$.

Further, the cost function only takes the next step $\mathcal{W}_k$ into consideration, and no prediction horizon of multiple consecutive $\{\mathcal{W}_{k+i}\}_{i \geq 0}$. This is done for simplicity, and since we are simply unsure how the model behaves when leaving the vicinity of $\mathcal{W}_{k-1}$. A prediction horizon could potentially be added together with a decreasing trust the further from $\mathcal{W}_{k-1}$ we move, to create an MPC-like problem formulation similar to [25, 26]. But the resulting optimization problem would be problematic, as it would become quite intricate with no guarantees on convexity.

4.1 Cost-optimizing controller

In each step, we will use the cost function (8) to decide the next W_k. As we consider no prediction horizon, and since the optimization problem is not convex with potentially multiple local minima, we will not try to optimize (8) until convergence in each time step. This would become unnecessarily costly, and only result in generating an optimal $\mathcal{W}_k$ given the step length constraint with no guarantees that it would actually move the system towards its global minimum.

Instead, a more direct approach is suitable. For demonstrative purposes, we use a very simple single gradient step to decide $\mathcal{W}_k$, based on the gradient of (8). Using automatic differentiation, this gradient ∇L_k can be directly obtained, despite the dependence on the two ODEs (1) and (4). Together with some constant scaling factor

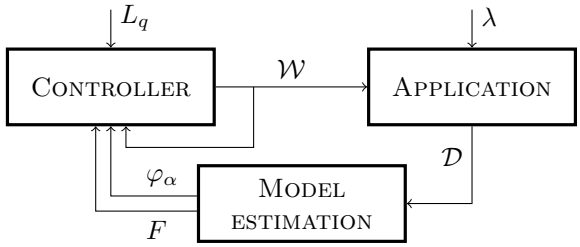


Figure 4: The controller sets some weights $\mathcal{W}$ for the application to run with. Data $\mathcal{D}$ is collected and used to fit model F as well as estimate the response time φ_α. These are then used by the controller to simulate the environment and update the old weights based on the gradient of the cost on the simulated state. Disturbances can act on both the controller, in the form of e.g. changes in the cost L_q, and on the application, in the form of e.g. changing workload λ.

α and a step size limiter c, we can then calculate the next $\mathcal{W}_k$ with the following gradient step update

$$\boldsymbol{w}^+ = \boldsymbol{w} - c\alpha\nabla_{\boldsymbol{w}} L_k\left(\mathcal{W}_{k-1}\right) \qquad \forall \boldsymbol{w} \in \mathcal{W}_{k-1} \tag{9}$$

where $\nabla_{\boldsymbol{w}}$ is the gradient w.r.t the elements in $\boldsymbol{w}$, and $\boldsymbol{w}^+$ the corresponding weight vector in $\mathcal{W}_k$. If (7) is not violated, then $c = 1$, otherwise it can be obtained as

$$c := \underset{c\in[0,1]}{\operatorname{argmin}} \sqrt{\sum_{\boldsymbol{w}\in\mathcal{W}_{k-1}} \left\|S(\boldsymbol{w}) - S\left[\boldsymbol{w}^+(c)\right]\right\|_2^2} = d\mathcal{W}_{\lim} \tag{10}$$

using e.g. bisection search.

The complete algorithm can thus be seen as an adaptable *gradient descent* scheme, where we after each step re-evaluate the model before calculating the gradient and deciding the next step to take. The resulting algorithm is summarized by the block diagram in Fig. 4 and the pseudocode in Algorithm 1.

Algorithm 1 Control and learning loop.

Algorithm 1a Run N iterations of data collection and parameter updates. Data collection is run for a duration h.

Initialize $\mathcal{W}_0$
for $k \leftarrow 1$ to N **do**
  Set $\mathcal{P}(\mathcal{W}_{k-1})$ as load balancing strategy
  $\mathcal{D}_k \leftarrow$ COLLECT_DATA(h)
  $F_k, \varphi_{.95} \leftarrow$ FIT$(\mathcal{D}_k)$
  $\mathcal{W}_k \leftarrow$ UPDATE$(\mathcal{W}_{k-1}, F_k, \varphi_{.95})$
end for

Algorithm 1b Calculate $\mathcal{W}_k$ based on $\mathcal{W}_{k-1}$ and data $\mathcal{D}_k$.

function UPDATE$(\mathcal{W}_{k-1}, F_k, \varphi_{.95})$
  $\mathcal{W}_k \leftarrow \emptyset$
  for $\boldsymbol{w} \in \mathcal{W}_{k-1}$ **do**
    $\nabla_{\boldsymbol{w}} L_k \leftarrow$ GRADIENT$(L_k(\boldsymbol{w} \mid F_k, \varphi_{.95}), \boldsymbol{w})$
    $\boldsymbol{w}^+ \leftarrow$ LIMITED_STEP$(\boldsymbol{w}, \nabla_{\boldsymbol{w}} L_k)$
    $\mathcal{W}_k \leftarrow \mathcal{W}_k \cup \{\boldsymbol{w}^+\}$
  end for
  return $\mathcal{W}_k$
end function

5 Evaluation

To test and showcase the online optimization algorithm, two experiments were performed on a small distributed microservice application.

5.1 Experimental setup

A similar setup to the one studied in [5] was considered, consisting of a simple microservice application deployed on a testbed of multiple Kubernetes clusters.

5.1.1 Kubernetes testbed

The federated application sandbox described in [28] was used as a testbed for the application. The sandbox consists of clusters of virtual machines deployed on OpenStack[5] in the Ericsson Research datacenter. To each cluster, 4 virtual machines, each with 4 vCPU and 4 Gb of RAM, were assigned. All virtual machines in each cluster are further connected via a cluster-specific isolated network. These networks are then connected to each other via a Gateway, which enables network characteristics between clusters to be easily emulated using TC netem[6]. On each cluster, Kubernetes[7] is then deployed along the service mesh Istio[8] to handle the application and to emulate a realistic cluster software stack. Istio further allows for easy extraction of the required tracing data for model fitting, and handling of cluster-to-cluster communication between microservices.

[5] https://www.openstack.org/
[6] https://man7.org/linux/man-pages/man8/tc-netem.8.html
[7] https://kubernetes.io/
[8] https://istio.io/

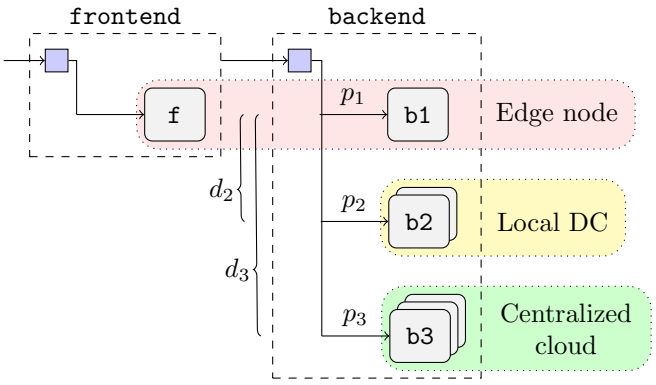


Figure 5: The incoming load passes through the frontend and is balanced over the replicas `bi` in the backend based on the probabilities p_i. The delay between the frontend and the replicas in the backend are d_i on average. The backends with lower delays are assumed to be more expensive to run, so a small edge node will be closer to the customer, but will likely have higher relative maintenance costs. This creates an optimization problem where delay and cost has to be balanced.

5.1.2 Application

A simplified version of the facedetection-as-a-service application shown in [5] was used as an example application. It consists of two services, a *frontend* implementing a user interface and image preprocessing, and a *backend* implementing a face-detection algorithm. Both services are implemented in Python using Flask[9] and Gunicorn[10]. We assume a structure as in Fig. 5 where the user-facing frontend only exists on the edge close to its users, and the backend is distributed across multiple sites, emulated by our different clusters. Each frontend-to-backend connection is associated with a delay d_i, and each backend is also associated with a computation cost C_i. We will assume that the higher costs are associated with lower delays, in order to create a trade-off between cost and latency. Such scenarios could occur in e.g. edge computing, where low-latency computations can be performed on the device or at geographically close edge data centers, but at an increased cost, while off-loading computations to larger sites is cheaper but subjected to longer communication delays.

5.1.3 Cost-optimizing controller

Using the load generator from [5], images are fed to the application as Poisson arrivals with rate λ. At every time step k, tracing data from Istio is collected to generate $\mathcal{D}_k$. The model F_k is fit to $\mathcal{D}_k$, and the cost function $L_k(\mathcal{W})$ is implemented using the Julia[11] programming language. Using `ForwardDiff.jl` [29] together with the ODE solver package `DifferentialEquations.jl` [30] allows for automatic differentiation of the cost function, so the gradient ∇L_k as well as the next $\mathcal{W}_k$ can be effortlessly calculated. The application, alongside the implementation of our online optimization algorithm, can be found here[12].

5.2 Two Backends - fixed steps in an offline experiment

In a first experiment, we consider the example application with only two replica sets of the backend, `b1` that is deployed on the same cluster as the frontend, and `b2` that is deployed on a different cluster. All connections between the two clusters are given a Pareto distributed additive delay with a 25ms mean, 5ms *jitter* (a TC netem term roughly equating standard deviation) and 25% correlation between samples. Hence, requests load balanced to `b2` will experience an additive delay.

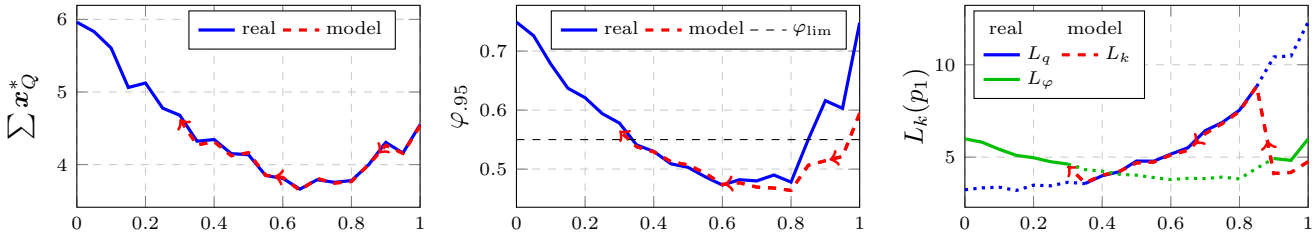
The probability constraint enables us to determine the load balancing probabilities directly by using a single parameter p_1, and then determining $p_2 = 1 - p_1$, removing the need for the weights and softmax function. This makes it feasible to collect data in a grid over $p_1 \in [0, 1]$, allowing us to run the agent offline against the recorded data with a fixed size step according to the grid that is in the direction of the gradient.

For the cost function, we let it be conditional on φ_α violating φ_{lim} for selecting either L_φ or L_q. L_φ is simply a scaled estimate of φ_α at stationarity, whereas L_q is 0 everywhere except at time t_f where it is a linear function of the state. We choose to look at the stationary values to make it easier to compare data.

$$L_k(p_1) = \begin{cases} C_\varphi \hat{\varphi}_\alpha(p_1) & \text{if } \varphi_\alpha > \varphi_{\text{lim}} \\ \boldsymbol{C}^T \boldsymbol{x}_Q(t_k + t_f \mid p_1) & \text{otherwise} \end{cases} \tag{11}$$

The conditional uses recorded data, as we like the constraint to be active if the real φ_α violates its limit. The actual cost is then implemented using a prediction from the model, $\hat{\varphi}_\alpha$, so it can be differentiated. We let $\alpha = 0.95$ to consider the 95th percentile of response times to the application, $\varphi_{\text{lim}} = 0.55$ms, $t_f = 5$s, and $\boldsymbol{C}$ be a zero vector except for two backends where it was set to $\boldsymbol{C}_b = [3, 1]$.

[9] `https://flask.palletsprojects.com/en/2.1.x/`
[10] `https://gunicorn.org/`
[11] https://julialang.org/
[12] TBD



(a) Total mean requests in application over p_1.

(b) 95th percentile of response time over p_1.

(c) Partial and full cost over p_1.

Figure 6: Results from the offline experiment with two backends. The values are plotted over the probability p_1. In 6(a) and 6(b) the blue lines show the value from recorded data $\mathcal{D}$, while the red dashed line shows the corresponding prediction from the fitted model. In 6(b) φ_{lim} shows when the cost (11) switches mode. The cost in turn can be seen in 6(c), where the blue and green lines correspond to the queue cost respective to the cost on overstepping the response-time threshold. The lines are filled where each cost is active, and dotted where they are inactive. The red dashed line shows the model's estimate of the full cost from both parts of the cost function.

We set $C_\varphi = 8$, though in this case it only affects plot scaling since the cost is conditional and the gradient step is fixed. For the model, we give each class in the queuing network 3 phase states, for a total size of $|\mathcal{S}| = 24$.

We record data for the offline experiment by creating a grid over p_1 with steps of 0.05 between 0 and 1. For each value of p_1, data is recorded for $h = 300$s using an arrival rate $\lambda = 14$. The optimization algorithm is then run against the recorded data, starting from $p_1 = 1$, stepping along the grid in the gradient direction. The results can be seen in Fig. 6. Fig. 6(a) shows the total mean requests present in the application, Fig. 6(b) shows the percentiles, and Fig. 6(c) shows the cost based on (11). A dashed red line shows the model prediction in all subfigures, while the blue (and green) lines represent values from recorded data. The cost in 6(c) is conditional on (11), where the recorded values are plotted for both individual costs while the model prediction is only for the combined function. The apparent noisiness can be attributed to the noisy data used for refitting the model.

As can be seen, the algorithm manages to step in the direction of a decreasing cost and ultimately find the minimum. Starting at $p_1 = 1$, the system is passing all load to `b1` resulting in $\varphi_{.95} > \varphi_{lim}$ and the cost is then based on the response time curve. Moving in the direction of $-d\varphi_{.95}/dp_1$ the response time as well as the total queue length is decreasing since we offload `b1` by routing some load to `b2` instead. Crossing the threshold φ_{lim}, the cost switch to L_q. Even though we see that the total queue length starts growing as p_1 goes below 0.65, the cost for `b1` is higher than `b2` and thus the queue based cost is still decreasing with decreasing p_1. Before reaching $p_1 = 0$ though, we hit the φ_{lim} threshold again, and here the simulation is stopped since the algorithm will start alternating between fixed steps in different directions.

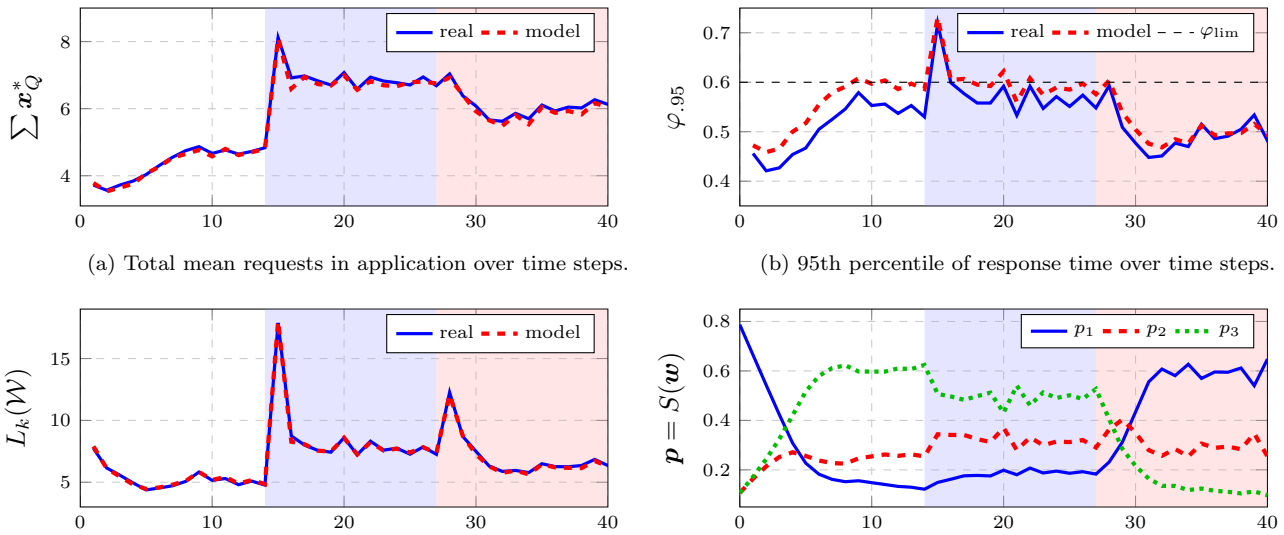
5.3 Three Backends - online optimization experiment

For the second experiment, we run everything live on the real application. We consider the same setup as in the first experiment, but also introduce a third backend replica (`b3`) deployed on a third cluster. All connections between the first and third clusters are given a Pareto distributed additive delay with a 50ms mean, 10ms jitter and 25% correlation between samples. As we now have more than 1 parameter to optimize over, we will resort to using the weight vectors in $\mathcal{W}$ as described in Section 4.

For the cost function, we again let L_q be a linear function of state at t_f and 0 for other times, to only consider stationary values and simplifying comparisons. However, the response-time constraint is included by setting L_φ as an exponential penalty function based on the difference between φ_α and φ_{lim}. Again, we base the activation of the constraint on recorded data, as we would like it to be active when the real φ_α is near or above its limit. The cost itself is based on the predicted $\hat{\varphi}_\alpha$ to make it differentiable. The resulting cost function becomes

$$\begin{aligned} L_k(\mathcal{W}) = \boldsymbol{C}^T \boldsymbol{x}_Q \left[t_k + t_f \mid S\left(\mathcal{W}\right)\right] \\ + C_\varphi e^{\mu(\varphi_{.95} - \varphi_{\text{lim}})} \hat{\varphi}_{.95} \left[S\left(\mathcal{W}\right)\right] \end{aligned} \tag{12}$$

where we set $\alpha = 0.95$, $\varphi_{\text{lim}} = 0.6$ms, $\mu = 10$, $C_\varphi = 5$, $t_f = 5$ms and $\boldsymbol{C}$ to a zero vector except for the backend replicas where it is set to $\boldsymbol{C}_b = [3, 2, 1]$. Further, the



(a) Total mean requests in application over time steps.

(b) 95th percentile of response time over time steps.

(c) Cost function over time steps.

(d) Load balancing probabilities over time steps.

Figure 7: Results from the online experiment with three backends. The values are plotted over the simulated time steps, each being one sampling period of 300 seconds. In 7(a), 7(b) and 7(c) the blue lines shows the value from recorded data $\mathcal{D}$, while the dashed red line shows the corresponding value from the fitted model. Further, in 7(d) the three lines corresponds to the three load balancing probabilities. Finally, the blue shaded area shows where the first disturbance is active on the arrival rate, while the shaded red area shows where the first and second disturbance is active on both arrival rate and queue length costs. A one time step lag can be seen on 7(a), 7(b), 7(c) compared to 7(d) as $\mathcal{D}_k$ is recorded using $\mathcal{W}_{k-1}$.

gradient step is given $\alpha = 0.5$ and $d\mathcal{W}_{\text{lim}} = 0.15$. For the model, we give each class in the queuing network 3 phase states, for a total size of $|\mathcal{S}| = 33$.

The system is loaded with Poisson arrivals at $\lambda = 15$ for a total of 40 time steps, with the initial weight vector is set to $\boldsymbol{w}_0 = [2, 0, 0]$, giving $\boldsymbol{p}_0 \approx [0.78, 0.11, 0.11]$. Each time step is given a duration of $h = 300$s. We further let the running system be influenced by two disturbances. The first disturbance is introduced at time step 14, where the arrival rate is suddenly increase by 50% to $\lambda = 22.5$. The second disturbance is introduced at time step 27, where the costs to the backend replicas are changed to $\boldsymbol{C}_b = [1, 2, 3]$.

The results from this experiment can be seen in Fig. 7. In all subfigures, the blue shaded area shows where the first disturbance is active, and the red area shows where both the first and second disturbances are active. Fig. 7(a) shows the total mean requests present in the application, Fig. 7(b) shows the response time percentiles, and Fig. 7(c) shows the cost based on (12). Both values from data (blue) and the corresponding fitted model (dashed red) are shown in these three subfigures. Finally, Fig. 7(d) shows the three load balancing probabilities from the frontend to `b1` (p_1, blue), to `b2` (p_2, dashed red) and to `b3` (p_3, dotted green).

As can be seen, the online optimization algorithm manages to drive the system towards a load balancing setting of less cost and counteract the disturbances in quite a few steps. At a few steps, it seems as the algorithm steps upwards in cost, but this can be attributed to noise. At first, the system shifts load from `b1` to `b2` and `b3` where costs are smaller. This decreases the cost, but also increases the total queue length and response time percentile, as `b2` and `b3` are associated with a higher site-to-site delay. The shift is mostly stopped when the percentile constraint is reached, but due to the simplicity of the gradient descent approach, the system experiences a slow final convergence. When the first disturbance in the form of a 50% increase in load is introduced, both the queue length and percentiles immediately increases. As the percentile constraint is now violated, the cost function spikes, which results in the gradient step aggressively moving the system back into a parameter configuration where the constraint is no longer violated. After the constraint is once again fulfilled, the cost function slowly settles to a minimum. At the activation of the second disturbance, in the

form of shifting the costs of `b1` and `b3`, the cost function increases while the queue length and percentile stays the same. The system then quickly shifts the load probabilities and reduce the cost until the effects of the penalty function becomes too large, and it settles to a slow final convergence. As can be seen in Fig. 7(b), there are at times rather large gaps remaining between the percentile and its limit. This has to do with the values assigned to the penalty function. Increasing C_φ and μ would lead to more aggressive handling of violations and a tighter gap, but they where kept fairly low to yield a more presentable cost function.

6 Conclusion and discussion

In this paper, we have demonstrated how automatic differentiation can be used to optimize a running distributed microservice application. This is done by deriving an online optimization scheme to minimize some holistic cost by tuning the probabilities of random load balancers between replica sets. Although not guaranteed to find the global cost minimum, the algorithm is shown to reduce cost while adhering to constraints on the response time percentile in an experimental evaluation. As the assumed microservice fluid model is fairly general, and the cost function can be arbitrarily defined, this online optimization scheme can be quickly adapted for a multitude of different load balancing scenarios.

Automatic differentiation allows us to differentiate functions whose derivatives would be too difficult to explicitly derive. In our case, it is used to differentiate through an arbitrary cost function L based on the solution of two dependent ODEs from our fluid model. Other models, e.g. the common mean-value analysis, could be used as well, but we chose the fluid model due to its transient values, validity for non-product-form networks and as it is quick to evaluate and differentiate using the Julia ecosystem. In our experiments, the model fitting and parameter update in Algorithm 1 takes about 10-15s, where the majority of the time goes to fitting the phase-type distributions using the EM algorithm, which can be made quicker by e.g. moment matching. The differentiation step itself is quick, in the range of 100ms.

Further, the generality of automatic differentiation allows us to consider a whole range of other ways of optimizing an application, e.g. scaling and migration, as long as such an action can be represented in e.g. the fluid model. Also, given an expression for the gradient and higher order partial derivatives, more advanced optimization methods can be used to decide how to update the control variables. In this paper we however chose to focus on optimizing load balancing using simple gradient stepping, to exemplify the procedure with a problem whose fluid model representation is simple, a comprehensible control variable update and a relatively easy experimental validation. More advanced decisions and optimization algorithms are left for future work.

The approach however has some drawbacks. First, there are no convergence guarantees and the performance is inherently dependent on the accuracy of the model. Identifying when a model is performing poorly is thus important to avoid driving the system into regions of high cost or constraint violations. Further, the generality of the approach is not only a boon. Deriving a suitable cost function and good optimization parameters can require both time and expert knowledge.

Acknowledgment

The two first authors contributed equally to this paper. The work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. All authors are members of the ELLIIT Excellence Center at Lund University. We also thank Ericsson Research for letting us run this in their datacenter.

References

[1] Tomas Cerny, Michael J. Donahoo, and Michal Trnka. Contextual understanding of microservice architecture: Current and future directions. *ACM SIGAPP Applied Computing Review*, 17(4):29–45, January 2018.

[2] Mor Balter. *Performance modeling and design of computer systems : queueing theory in action*. Cambridge University Press, Cambridge, 2013.

[3] Danilo Ardagna, Giuliano Casale, Michele Ciavotta, Juan F. Pérez, and Weikun Wang. Quality-of-service in cloud computing: modeling techniques and their applications. *Journal of Internet Services and Applications*, 5(1):11, September 2014.

[4] Gunter Bolch, Stefan Greiner, Hermann de Meer, and Kishor S. Trivedi. *Queueing Networks and*

Markov Chains. John Wiley & Sons, Inc., Hoboken, N.J, March 2006.

[5] Johan Ruuskanen and Anton Cervin. Distributed online extraction of a fluid model for microservice applications using local tracing data. *To appear in IEEE CLOUD 2022. Email johan.ruuskanen@control.lth.se for early access.*

[6] Søren Asmussen, Olle Nerman, and Marita Olsson. Fitting phase-type distributions via the em algorithm. *Scandinavian J. Statistics*, 23(4):419–441, 1996.

[7] Johan Ruuskanen, Tommi Berner, Karl-Erik Årzén, and Anton Cervin. Improving the mean-field fluid model of processor sharing queueing networks for dynamic performance models in cloud computing. *Performance Evaluation*, 151:102231, 2021.

[8] M. S. P. Eastham. 2968. On the Definition of Dual Numbers. *The Mathematical Gazette*, 45(353):232–233, 1961.

[9] Louis B Rall. *Automatic differentiation: Techniques and applications*. Springer, 1981.

[10] Yung-Terng Wang and Morris. Load sharing in distributed systems. *IEEE Transactions on Computers*, C-34(3):204–217, 1985.

[11] Sandeep Sharma, Sarabjit Singh, and Meenakshi Sharma. Performance Analysis of Load Balancing Algorithms. page 4, 2008.

[12] Rich Lee and Bingchiang Jeng. Load-balancing tactics in cloud. In *2011 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, pages 447–454. IEEE, 2011.

[13] Varun Gupta, Mor Harchol Balter, Karl Sigman, and Ward Whitt. Analysis of join-the-shortest-queue routing for web server farms. *Performance Evaluation*, 64(9):1062–1081, 2007. Performance 2007.

[14] Yossi Azar, Andrei Z Broder, Anna R Karlin, and Eli Upfal. Balanced allocations. In *Proceedings of the twenty-sixth annual ACM symposium on theory of computing*, pages 593–602, 1994.

[15] Maury Bramson, Yi Lu, and Balaji Prabhakar. Randomized load balancing with general service time distributions. SIGMETRICS '10, page 275–286, New York, NY, USA, 2010. Association for Computing Machinery.

[16] Yi Lu, Qiaomin Xie, Gabriel Kliot, Alan Geller, James R. Larus, and Albert Greenberg. Join-idle-queue: A novel load balancing algorithm for dynamically scalable web services. *Performance Evaluation*, 68(11):1056–1071, 2011. Special Issue: Performance 2011.

[17] Sebastiano Spicuglia, Lydia Y Chen, and Walter Binder. Join the best queue: Reducing performance variability in heterogeneous systems. In *2013 IEEE Sixth International Conference on Cloud Computing*, pages 139–146. IEEE, 2013.

[18] Luigi Fratta, Mario Gerla, and Leonard Kleinrock. The flow deviation method: An approach to store-and-forward communication network design. *Networks*, 3(2):97–133, 1973.

[19] Sem C Borst. Optimal probabilistic allocation of customer types to servers. *ACM SIGMETRICS Performance Evaluation Review*, 23(1):116–125, 1995.

[20] Xin Guo, Yingdong Lu, and Mark S Squillante. Optimal probabilistic routing in distributed parallel queues. *PERFORMANCE EVALUATION REVIEW*, 32(2):53–54, 2004.

[21] Hiroshi Kobayashi and Mario Gerla. Optimal routing in closed queuing networks. *ACM Transactions on Computer Systems (TOCS)*, 1(4):294–310, 1983.

[22] Jonatha Anselmi and Giuliano Casale. Heavy-traffic revenue maximization in parallel multiclass queues. *Performance Evaluation*, 70(10):806–821, 2013. Proceedings of IFIP Performance 2013 Conference.

[23] Weikun Wang and Giuliano Casale. Evaluating weighted round robin load balancing for cloud web services. In *2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 393–400, 2014.

[24] A Hordijk and JA Loeve. Optimal static customer routing in a closed queuing network. *Statistica Neerlandica*, 54(2):148–159, 2000.

[25] Emilio Incerto, Mirco Tribastone, and Catia Trubiani. Software performance self-adaptation through efficient model predictive control. In *2017 32nd IEEE/ACM International Conference on Automated*

Software Engineering (ASE), pages 485–496, October 2017.

[26] Emilio Incerto, Mirco Tribastone, and Catia Trubiani. Combined vertical and horizontal autoscaling through model predictive control. In Marco Aldinucci, Luca Padovani, and Massimo Torquati, editors, *Euro-Par 2018: Parallel Processing*, pages 147–159, Cham, 2018. Springer International Publishing.

[27] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[28] Johan Ruuskanen, Haorui Peng, Alfred Åkesson, Lars Larsson, and Maria Kihl. Fedapp: a research sandbox for application orchestration in federated clouds using openstack, 2021.

[29] Jarrett Revels, Miles Lubin, and Theodore Papamarkou. Forward-Mode Automatic Differentiation in Julia. *arXiv:1607.07892 [cs]*, July 2016.

[30] Christopher Rackauckas and Qing Nie. Differentialequations.jl–a performant and feature-rich ecosystem for solving differential equations in julia. *Journal of Open Research Software*, 5(1):p.15, 2017.