Web Scaling Frameworks for Web Services in the Cloud

Thomas Fankhauser, *Student Member, IEEE,* Qi Wang, *Member, IEEE,* Ansgar Gerlicher, *Member, IEEE,* Christos Grecos, *Senior Member, IEEE,* and Xinheng Wang, *Senior Member, IEEE*

Abstract—Nowadays, web services have to accommodate a significant and ever-increasing number of requests due to high interactivity of current applications. Although the built-in elasticity offered by a cloud can mitigate this challenge, it is highly desirable that applications can be built in a scalable fashion. State-of-the-art Web Application Frameworks (WAFs) focus on the creation of application logic and do not offer integrated cloud scaling concepts. As the creation of such scaling systems is very complex, we proposed in our recent work the concept of Web Scaling Frameworks (WSFs) in order to offload scaling to another layer of abstraction. In this work, a detailed design for WSFs including necessary modules, interfaces and components is presented. A mathematical model used for performance rating is evaluated and enhanced on a computing cluster of 42 machines. Traffic traces from over 25 million real-world applications are analysed and evaluated on the cluster to compare the WSF performance with a traditional scaling approach. The results show that the application of WSFs can substantially reduce the number of total machines needed for three representative real-world applications — a social network, a trip planner and the FIFA World Cup 98 website — by 32%, 63% and 92%, respectively.

Index Terms—scalability, web service, cloud computing, web scaling frameworks, performance evaluation, software architecture

1 Introduction

The demands for modern scalable web services are increasing rapidly due to the soaring social nature of the web and the upsurge of the total number of mobile devices. Web services have to deal with a high level of interactivity in applications, which in turn introduces enormous amounts of requests. Static websites are replaced by dynamic and highly interactive applications. For instance, TV shows deploy apps that allow users to influence the course of the show, advertisements are brought to customers only if they remain in the vicinity of advertised target locations, smart sensors deliver data for all kinds of metrics which users see on their mobile devices, and cars communicate traffic situations, report traffic jams and find intelligent routes based on live data.

1.1 Motivation

The above scenarios introduce new challenges to web service providers and developers. As single-server systems are not able to handle the increased load, applications need to be built in a scalable fashion. Requests have to be balanced over all available machines, resources need to be shared without conflicting versions, distributed transactions have to be processed in a fault tolerant manner, and the number of machines has to be adapted to highly dynamic traffic situations. Typically, web service providers need to reach a critical mass of users to be commercially sustainable. If the critical mass is reached, the web services need to be able to scale up immediately to stay in business. Before this threshold is reached, providers need to focus on the business logic, which often prevents detailed scalability considerations. State-of-the-art Web Application Frameworks (WAFs) are designed to abstract common functionalities needed for the efficient implementation of web services. They focus on the creation of application logic, data structures, data validation, view layer presentation and session handling. They do not offer integrated scaling concepts that handle the provisioning of resources, employ optimised caching strategies or ensure guaranteed response times. Today, web service providers have to create custom-built systems that consider these scalability issues manually.

1.2 Background

As the creation of such scaling systems is a very complex matter, we proposed Web Scaling Frameworks (WSFs) in our recent work [1]. We proposed that WSFs take over the responsibilities of scaling by embedding existing WAFs in a larger system. In [1] we have shown empirically that the application of a WSF can triple the request throughput performance of a single machine. We were also able to show analytically that we can reduce the number of total machines by 44% in certain cases. However, the model was not evaluated on multiple machines nor with real-world traffic traces. Moreover, the traditional app version in the evaluation did not use a cache and the analytical model was not fully developed. In addition, a detailed design of WSFs was not given.

1.3 Contribution

In this work, we specify a detailed design of WSFs including necessary modules, interfaces and the composition of components, where modules define the core functionalities of a

- *T. Fankhauser, Q. Wang and X. Wang are with the School of Computing, University of the West of Scotland*
  E-mail: {Thomas.Fankhauser, Qi.Wang, Xinheng.Wang}@uws.ac.uk
- *C. Grecos is an Independent Imaging Consultant.*
  E-mail: grecoschristos@gmail.com
- *A. Gerlicher and T. Fankhauser are with the Stuttgart Media University.*
  E-mail: {gerlicher, fankhauser}@hdm-stuttgart.de

TABLE 1
Categorisation of Related Work

Scalability (2.1)	**References**
Framework or Platform	this, [2], [3], [4]
Modelling	this, [4], [5], [6], [7], [8]
Elastic Computing	[4], [5], [6], [7], [8], [9], [10]
Caching (2.2)	**References**
Strategy	this, [9], [11], [12], [13]
Policies	this, [9], [10]
Architecture (2.3)	**References**
Pattern	this, [14], [15], [16], [17], [18]
Methodology	this, [14], [15], [16], [17], [18], [19], [20]
PaaS Model	this, [2], [14]
SaaS Model	[5], [14]

WSF and are connected to components through interfaces. Firstly, we significantly enhance our mathematical model presented in [1] and evaluate it on a cloud computing cluster. The network delay is remodeled using a linear or quadratic form, which improves the accuracy and eliminates a parameter that had to be estimated from a series of elaborate tests in [1]. The linear total machines regression is generalised for the WSF and WAF-only version, which allows approximating the performance of both versions for comparison. A performance-concurrency-width triplet is composed that introduces a new way for comparing performance between components and an equation for the break-even point of the post-processing delay is presented, which allows estimating the available time for post-processing. Secondly, for improved web service scalability, we highlight a set of optimisation schemes including a caching and post-processing strategy, a novel request flow routing mechanism, an optimal concurrency range calculation algorithm, and optimisations for highly dynamic content. Finally, in the evaluation, we provide both application versions with caches to represent real scenarios more precisely. We evaluate an extended version of the model with 42 machines and assess the performance using parameters extracted from a total of 25 million trip planner, social network and soccer worldcup traces.

The remainder of the paper is organised as follows: Section 2 summarises the results of the literature review of related work. Section 3 illustrates the general concept of WSFs with modules, interfaces and components. Section 4 derives a concrete prototype of a WSF, whilst Section 5 develops the models for the prototype and Section 6 evaluates the models with a cloud cluster. Section 7 outlines the results and conclusions and gives a perspective on future research.

2 Related Work

We reviewed and classified related work into three categories (Table 1) defined by the design goals of our proposed WSFs: Work in the *Scalability* category deals with the creation and modelling of scalable frameworks and platforms. The *Caching* category deals with eviction methods and work related to our caching strategy and the *Cloud Architecture Patterns* category deals with work proposing general cloud taxonomies, architectures and composition patterns.

2.1 Scalability

2.1.1 Platforms and Frameworks

The AppScale Cloud Platform [2] is a distributed software system that implements a Platform-as-a-Service (PaaS) that allows deployment of cloud applications. The TwoSpot [3] PaaS enables hosting multiple, sandboxed Java compatible applications and has a focus on the prevention of vendor lock-in. Unlike the WSF we propose, both platforms do not suggest a caching strategy or service composition architecture, where the performance of each service (component) can be calculated with respect to a targeted load.

2.1.2 Modelling

The work in [5] establishes a formal measure for under- and over-provisioning of virtualised resources in cloud infrastructures specifically for Software-as-a-Service (SaaS) platform deployments and proposes new resource allocation mechanisms based on tenant isolation, VM instance allocation and load balancing. The proposed mechanisms are specifically optimised for SaaSs, whereas our proposed WSF is a PaaS middleware [14] that implements a general web service framework. An application that is developed using our proposed WSF does not necessarily have multiple tenants (customers). Thus, *itesm-cloud*'s [5] balancing of the Virtual Machine (VM) load that is based on the tenants of a SaaS platform is not applicable for our proposed framework. In [6], the authors propose an optimal VM-level auto-scaling scheme with cost-latency trade-off. The scheme predicts the number of requests based on history data and then gives instructions for service provisioning, but does not propose a service composition architecture or caching strategy. In [4], the authors model an elastic scaling approach that makes use of cost-aware criteria to detect and analyse the bottlenecks within multi-tier cloud-based applications. The approach is based on monitors that measure the current workload and scale up or down based on the performance, however it does not employ a caching strategy or application performance profile for optimal load targeting we propose in Section 5.

2.1.3 Elastic Computing

The authors in [8] provide an overview of the key concepts of stream processing in databases, with special focus on adaptivity and cloud-based elasticity. The processing mechanisms we propose in this work (Section 4) operate on a stream of events (requests) that are processed by a processing agent (worker) and delivered to a client. The work [8] however does not employ a caching strategy, performance model or special composition of components. In [7], the authors introduce novel ideas on testing for elastic computing systems, such as impact, fatigue and shear testing. The auto-scaling capabilities needed for such elastic systems are part of the self-adaptive systems field where [18] introduces an architectural blueprint for autonomic computing. The WSF we propose does not employ algorithms for elastic auto-scaling, though this is subject to future work.

2.2 Caching

2.2.1 Strategies

The authors in [9] design and evaluate a caching policy that minimises the cost of a cloud-based system by taking into

account the frequency of consumption of an item and the cloud cost model. As prices for cloud storage are expected to continue to drop [11], [12], we expect that the storage of resources will be cheaper than the recurrent processing on a cache miss. Hence, in this work we propose to fully utilise caching and thereby maximise scalability, performance predictability and response times.

2.2.2 Policies

The eviction of cache objects is performed to minimise the cache size and cost. It is either based on *timeouts* [9], *access frequency* [9] or *access patterns* [10] that are based on machine-learning algorithms. One major issue of the eviction is that on arrival of a request it is uncertain if the requested resource is cached (hit) or needs to be recomputed (miss). Thus, provisioning algorithms need to employ the concept of a hit-miss probability to calculate performance and can not guarantee response times for cacheable resources. Due to the fast decreasing in the price of caches, we propose to cache all available resources and abandon cache eviction. This allows strictly isolating cached and uncached requests in the performance model yet requires a special post-processing of requests to be outlined in Section 4.

2.3 Cloud Architecture Patterns

In their composite cloud patterns [14], the authors propose a *two-tier* and *three-tier* application pattern. The composition we propose in this work engages with requests before they arrive at an application, hence applications can implement both patterns with a modified request flow. A detailed explanation of the differences is given in Section 4. The microservice architectural style [15], [16] is an approach to developing a single application as a suite of small services that communicate through lightweight mechanisms such as an HTTP resource API. Our proposed WSF is designed to compose multiple small services into a single web application, so it encourages the use of the microservice architecture pattern. The Command Query Responsibility Segregation (CQRS) [17], [20] is a pattern to split the conceptual representation of a domain into separate models for update and display. In this work we follow the pattern by creating separate request flow sub-systems for queries and commands. Further details are given in Section 4.

3 Conceptual Architecture

The WSF architecture we propose is an abstract concept that is valid for all implementations of WSFs. In the subsequent section we give an example for an implementation with a proposed prototype.

3.1 Modules

As shown in Fig. 1 *(a)*, a WSF is a program that is hosted by a cloud-provider in production, or on a local machine for development. Our requirements specification divides the framework-functionalities into six modules, which are listed in Table 2 for overview.

Together with the *provision interface*, the *provision module* M_P implements the *provider adapter* cloud application pattern [14]. The module is concerned with the management of cloud-provided components that includes the execution of auto-scaling and component composition. To simplify provisioning, we propose compliance to TOSCA [21] or the deployment of Docker containers [22] to cloud-provided docker runtimes that are offered by multiple cloud providers including Amazon's Container Service ECS [23] and Google's Container Service [24].

In order to monitor and calculate the current overall system performance, the *metrics module* M_M along with the *metrics interface* is responsible for collecting performance parameters from the components. Existing monitoring solutions exist, although they are vendor-specific as e.g. Amazon's CloudWatch [23] or Google's Cloud Monitor [24]. Hence, we propose to implement a custom metrics module until a standardised monitoring approach emerges. The performance model we develop in this work is solely based on delays that are influenced by the network, processing time, request size and concurrency for each component. We propose to collect these *component-parameters* using a second graph of components, the *metrics components*, which is completely decoupled from the graph of components that processes the production requests. The metrics module frequently executes a series of tests against this metrics components to determine the current values for each of the component-parameters. The metrics interface details exactly the component-parameters that are highlighted explicitly in the modelling section.

Each collection of metrics for each component is stored

TABLE 2
Conceptual Framework Modules, Interfaces and Parameters

Module	**Description**
M_P: Provision	Manages the component topology and provisions machines on different cloud-providers.
M_M: Metrics	Measures the performance of each component as input to the performance model.
M_S: Storage	Stores the performance metrics for scaling decisions and parameters.
M_L: Logic	Decides provisioning of components based on performance metrics and parameters.
M_I: Interface	A graphical user interface or API to manage parameters.
Interface	**Description**
I_C: **Component**	
Provision	Describes how the component topology is created and maintained.
Metric	Details what metrics a component needs to report in order to be scaled.
I_A: **Application**	
HTTP	The application is required to respond to HTTP requests.
Resources	Enlists the methods an application needs to keep its resources up-to-date in the cache.
Parameters	**Description**
Component	Are metrics collected for each component and used for provisioning.
System	Parameters that affect the system wide scaling algorithms.
Traffic	Describe the performance goals of the overall system.

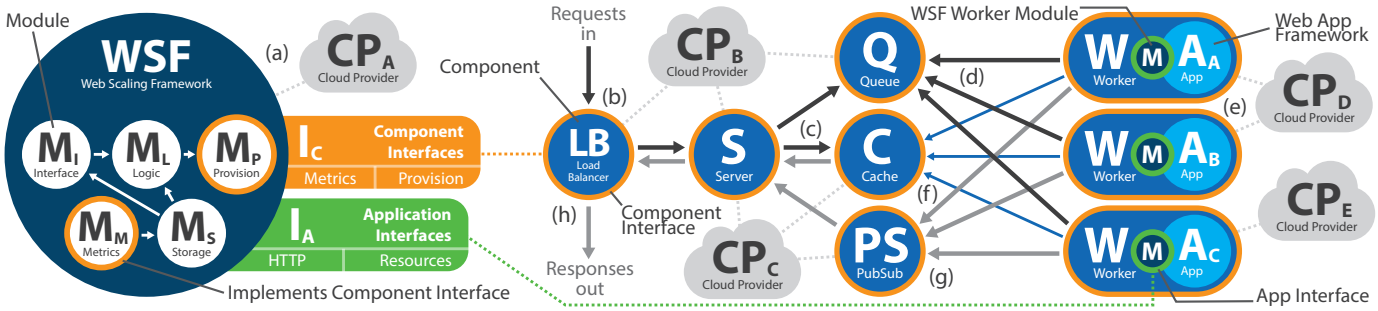

Fig. 1. Architecture overview of the proposed WSF that manages multiple components and applications hosted by different cloud providers. (a) illustrates the modules and interfaces of a WSF where the white arrows denote the data flow. (b-h) show the flow of a request through the components where black is the request and grey is the response. (e) highlights the embedding of applications into worker components for processing. (f) illustrates the resource update from the worker to the cache where the blue arrows denote the data flow. The novel routing scheme splits traffic at (c) into processing sub-system (LB,S,Q,W,A,W,PS,S,LB) and read sub-system (LB,S,C,S,LB).

in the *storage module* M_S which can be a database of any form.

The *logic module* M_L is responsible for scaling decisions and thereby implements the *elasticity management process* cloud application pattern [14]. Based on metrics stored in the storage module and *system-parameters* that e.g. define the aggressiveness of auto-scaling, the logic module communicates its decisions to the provision module for execution.

The *interface module* M_I is an optional module we propose to provide feedback and control to the user. It can offer the component-parameters and system-parameters as a graphical user interface (admin web-console) or provide an API for machine-controlled access.

To compose multiple applications that are orchestrated by a WSF, the *worker module* M_W (Figure. 1 *(e)*) is responsible for processing requests with the help of an application. Based on metrics from the storage module and metrics the worker collects for its embedded application, it handles the request throttling, resolves and executes the updates of dependencies and pushes content to the cache.

3.2 Interfaces

Figure. 1 *(a)* highlights how a WSF is connected to components and applications it manages.

We propose to define *component* and *application interfaces* as communication abstractions. In Table 2 and the previous section we highlighted the *provision* and *metrics interface* for components as they are tightly coupled with their corresponding modules.

Application interfaces define how a WSF communicates with its applications or microservices [15] that are created using a WAF. We propose to use standard HTTP requests for communication, as every WAF used to create an application has to support HTTP by the definition of a web service.

In order to decouple the cache access and resource management from the application, we propose the worker module to implement a *resources interface*. The application can use this interface to communicate with the worker module. For the caching strategy we propose in our prototype, the resources interface consists of the following methods:

- **pushCache(keys, values):** Pushes resource contents as values to the respective keys in the cache
- **deleteCache(keys):** Deletes cache resources by their keys
- **syncUpdate(keys):** Makes the worker request the specified keys before the current response is returned
- **asyncUpdate(keys):** Makes the worker request the specified keys without timing constrains

3.3 Parameters

We propose to categorise the parameters that describe the state and influence the behaviour of all WSFs into *component-*, *system-* and *traffic-parameters* as illustrated in Table 2. Component-parameters describe the metrics that are collected for each component, respectively. In the prototype we present in this work, the parameters are the different delays needed to calculate the performance of each component and are described in the metrics interface. System-parameters influence the overall behaviour of a WSF and are used as input to scaling-algorithms. They can influence the sensitivity of scaling, ensure the compliance to a maximum budget or set constraints regarding the service-level agreements. However, the prototype we propose does not currently employ any system-specific parameters as this is subject to future work. Traffic-parameters characterise the performance goal of a system, such as the total requests it can handle per second, the maximum number of concurrent requests or the maximum time that can pass between a request and a response. In this work, we focus on the total requests a system can handle per second and the maximum number of concurrent users.

3.4 Components

Components are individually scalable services from cloud offerings [14], or self-managed services running on Infrastructure-as-a-Service (IaaS). In Fig. 1 *(b-h)*, we illustrate the components we use for our prototype. We propose these components as a minimal set of components based on recent research on cloud application patterns [14]. Using an exemplary provisioning setup with Docker containers [22], a component is specified by a *Dockerfile* which describes the build process for the container including the operating system, internal environment and applications such as a cache, worker or queue. The provision module can easily execute scaling by launching multiple containers in their respective container service.

3.5 Request Flow Routing

The composition of components and routing of requests is specific to each WSF. Following the Docker example, containers can be linked to form a topology for component composition and routing. One major design goal for WSFs is to implement efficient routing of requests. In Section 4 we propose a novel request flow scheme that optimises performance and enhances scalability by minimising the request flow graph for every request.

3.6 Caching Strategies

The efficient routing of requests is further supported by efficient caching strategies in WSFs. A WSF offers a caching interface (Section 3.2) to all of its WAF applications and manages cache eviction, scaling and the processing of cache resource dependencies. In Section 4.3 we propose to abandon cache eviction based on policies [9], [10] by placing every requestable resource in the cache and apply smart post-processing to manage cache resources and its dependencies.

3.7 Processing Strategies

The processing of requests using a WSF is performed by multiple applications. Resources managed by these applications can depend on each other, e.g., a sitemap of a blog depends on the posts of a blog. To keep the depending resources in sync, additional post-processing is required. In Section 4 we propose a new post-processing strategy that minimises response times while maintaining scalable eventual consistency.

3.8 Performance Profiling

The request throughput a WAF is able to deliver is heavily dependent on the concurrency of the incoming requests and has an optimal concurrency range (Section 5.5) where the throughput is at its maximum. In our prototype, we propose a new algorithm to frequently profile the performance of a WAF component and thus enable the WSF to adaptively keep the concurrency in this measured optimal concurrency range for optimal throughput.

4 Prototype Implementation

To examine the feasibility and performance of the abstract WSF concept, we implement a prototype with a novel composition of components that is able to calculate and optimise the overall throughput of a web service. The design goal of the proposed prototype is to achieve improved scalability and performance compared with contemporary WAFs. We model the request throughput and scalability for analytical evaluation and validate our model using multiple implementations on a cloud cluster of 42 machines. In order to compare our prototype with a traditional scaling approach, we further evaluate the performance of both approaches with three real-world traffic traces.

4.1 Background

As a first step, the current state-of-the art cloud offerings [14] that can be used as components are examined. The authors in [14] distinguish between *processing*, *storage* and *communication offerings*. Processing offerings execute work in specialised execution environments or on virtualised hardware. Typical components are servers that process requests with the help of applications that are implemented using a WAF. Storage offerings store data with various requirements where key-value caches, relational databases, block and blob storage components are typical examples. Communication offerings connect processing and storage offerings. Typical components include load-balancers, queues, publish-subscribe systems and virtual networking.

The normal composition of components to a native cloud application implements either a *two-tier* or *three-tier* cloud application pattern, or is a *content distribution network* [14]. The two-tier and three-tier patterns are similar, where the three-tier pattern decouples the presentation and business logic tier into separate tiers.

4.2 Normal Application Version V_N

As a reference architecture we compose a *Normal Application Version* V_N from the two-tier pattern [14], where as components we use a load-balancer LB, an application A for the presentation and business logic tier and a cache C for the data tier. The application component A runs an implementation of a web application that is created using a WAF. The processing of requests follows the graph with the edges $S_N = \{(LB, A), (A, C), (C, A), (A, LB)\}$. We use this normal version V_N to compare our prototype with the traditional scaling approach throughout this work.

4.3 Scaled Application Version V_S

The prototype we propose in this work uses more components and a different composition than the normal version V_N. It implements a *Scaled Application Version* V_S that uses a WSF in combination with a WAF. Fig. 1 *(b-h)* illustrates the components our prototype adds to the normal version V_N: A request queue Q, a server S, a publish-subscribe component PS and a worker component W that embeds an implementation of a web application A that is created using a WAF.

4.3.1 Request Flow

We create the illustrated composition of components (Fig. 1 *(b-h)*) with two major design goals: optimised performance and enhanced scalability.

Our approach to optimising the performance is to minimise the request flow graph for every request. We do this by segregating commands from queries [17], [20], where the segregation is performed based on the HTTP-verb of a request. The goal is to minimise the components visited by a request and reduce the load for the app A which executes expensive request processing.

Fig. 1 *(b-h)* illustrates the detailed flow of a request through the components: Requests enter the system through multiple load-balancers LB (Fig. 1 *(b)*). The load-balancers LB forward the request to one of the servers S. The server

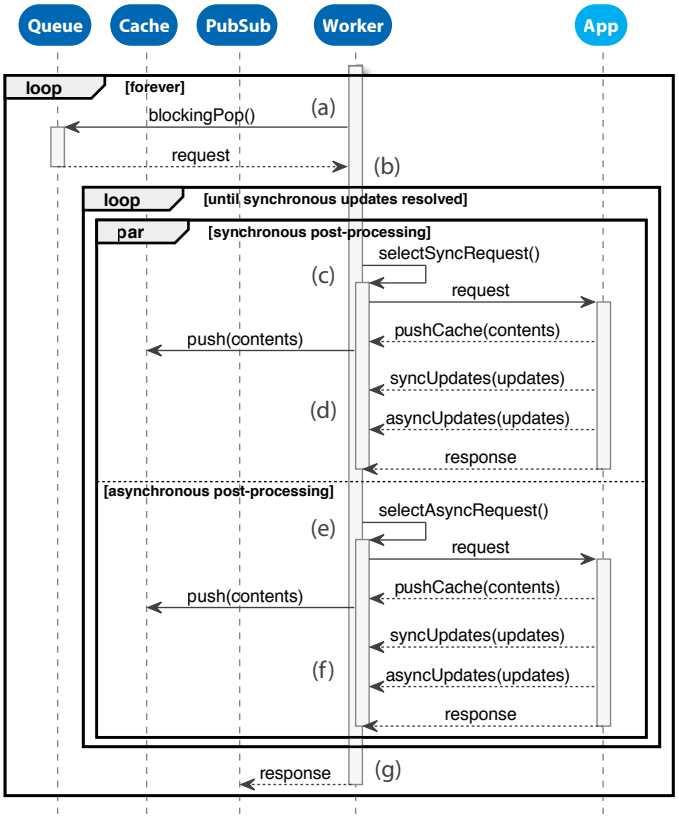

Fig. 2. Request processing and post-processing controlled by the worker. (a) illustrates the pop of a request by the worker from the queue. In (b) the processing and post-processing is executed until all synchronous updates are processed. (c,e) highlights the parallel selection for processing of a synchronous and asynchronous request. In (d,e) each of the requests pushes content to the cache and declares its update dependencies. (g) returns the response to the initial request at (a).

S now decides whether the request is a Read-Request (R_R) or a Processing-Request (R_P). R_Rs are routed to the Read Sub-System (SR) which is a graph with the directed edges $SR = \{(LB, S), (S, C), (C, S), (S, LB)\}$ (Fig. 1 *(b,c,h)*). R_Ps are routed to the Processing Sub-System (SP) where $SP = \{(LB, S), (S, Q), (Q, W), (W, A), (A, W), (W, PS), (PS, S), (S, LB)\}$ (Fig. 1 *(b-h)*). The routing is conducted based on the HTTP-verb where all requests with the verb *GET* or *HEAD* are routed to the read sub-system SR and all others to the processing sub-system SP. This is only possible as by definition all deliverable resources are stored in the cache C. Therefore, for a read-request R_R the server S looks up the cache key including possible fragments and delivers the response. Processing-requests R_Ps always need to be processed, so the server S puts them in the queue Q and listens for the response at the publish-subscribe system PS. A worker W with free resources pops the request from the queue Q. The app A processes the request and the worker W publishes the response to the publish-subscribe system PS where the server S is waiting for it. Finally the server S delivers the response back to the client (Fig. 1 *(h)*).

Our approach to enhancing the scalability is based on the read and processing sub-systems and the component structure. Depending on the workload, both sub-systems can be scaled independently on a component level. Additionally, a component exposes its scalable machines as nodes. In our prototype, we create multiple decoupled and parallel graphs of nodes, where on arrival each request is designated to one of the graphs. With this approach, we try to minimise the influence requests can have on each other as they are flowing in parallel.

TABLE 3
Equations of the analytical prototype evaluation alongside the modifications of the model presented in our previous work [1].

Component Model	**Modification to [1]**
Processing Delay	Extracted Request Size Delay and added Post-Processing Delay
Request Size Delay	Explicit Definition
Network Delay	Extracted linear and quadratic form
Maximum Request Flow	Added Machine parameter
Machines for Target Flow	New
Composition Model	**Modification to [1]**
Maximum Request Flow	New
Machines for Target Flow	Generalised form
Linear Machines Regression for Target Flow	Generalised form
Performance Comparison	**Modification to [1]**
Relative Average Machine Reduction	Removed percentage
Break-Even Point for Post-Processing	New
Performance Optimisation	**Modification to [1]**
Optimal Concurrency Range for Component	New

4.3.2 *Processing and Post-Processing*

In order to use the optimised request flow we propose, all deliverable resources need to initially be put and kept up-to-date in the cache C. To initially fill up the cache, we keep an index of all available resources that are cached before the system goes into operating state. To keep the cache up-to-date, the prototype uses the processing strategy illustrated in the sequence diagram in Fig. 2. The server S puts requests that need processing into the queue Q. If a worker has available resource to process a request, it pops a request from the queue Fig. 2 *(a)* and declares it a *synchronous update* Fig. 2 *(b)*. In the parallel processing section Fig. 2 *(c-f)*, the worker W selects a request that needs processing by the app A (Fig. 2 *(c,e)*) and initiates the request. The app then processes the request, pushes updated cache contents and declares its synchronous and asynchronous updates at Fig. 2 *(d,f)* before it returns the response to the worker. Once all declared synchronous updates are processed, the response to the initial request is returned Fig. 2 *(g)*.

A general goal to minimise the response time for a request is to keep the number of synchronous updates as low as possible as all of them are processed before the response to the initial request is sent out. The asynchronous dependencies are decoupled from the original request and processed when the system has available resources. The proposed algorithm chooses eventual consistency over strong consistency in the cache [17]. During the processing step, the resource that is delivered by the cache might not be updated yet, however eventually it will be.

TABLE 4
Component parameters that are used to describe and model the performance of a single component x.

Goal-oriented	**Description**
Request-Flow / Second: $f_x \in [0, \infty]$	The requests that flow through a component in one second.
Target Flow / Second: $t_x \in [0, \infty]$	A desired target f_x for a component.
Performance-based	**Description**
Network Delay: $d_{n,x} \in [0, \infty]$ in s	The time it takes a request to travel the full network stack.
Network Delay Gain: $d_{g,x} \in [0, \infty]$ in s	The linear or quadratic time factor by which $d_{n,x}$ increases.
Lookup Delay (V_N): $d_{l,x} \in [0, \infty]$ in s	The time it takes the app A to lookup a resource in the cache C.
Processing Delay: $d_{p,x} \in [0, \infty]$ in s	The time it takes a component to process a request.
Post-Processing Delay: $d_{pp,x} \in [0, \infty]$	The time needed to post-process a request.
Size Delay: $d_{s,x} \in [0, \infty]$ in s	The time a single kilobyte adds to the delay.
Workload-based	**Description**
Concurrent Users: $c_x \in [1, \infty]$	The number of concurrent users or connections.
Size: $s_x \in [0, \infty]$ in kB	The size of a single request-response round-trip.
Deployment-based	**Description**
Machines: $m_x \in [1, \infty]$	The number of machines that are used for one component.

5 Analytical Prototype Modelling

To be able to analytically evaluate our proposed prototype we develop mathematical performance models for a component (Section 5.2), the composition of multiple components (Section 5.3), the performance comparison of two compositions of components (Section 5.4), and the performance optimisation of a component (Section 5.5). An overview of the developed equations alongside their relation to our previous work [1] can be found in Table 3.

5.1 Performance Goals

The general performance goals of the modelled prototype are either the reduction of the number of total machines M needed to satisfy a desired target request flow F or in reverse the increase of the request flow F with a given number of total machines M. In this evaluation, we model two compositions of components we described in the previous section: The normal application version V_N and the scaled application version V_S. For our proposed prototype we aim to achieve that $F_S > F_N$ with equal M or $M_S < M_N$ with equal F.

5.2 Component Models

As a first step we model the performance of a single component that later is composed to a larger system.

5.2.1 Parameters

Each component has its own set of parameters denoted in Table 4. Component parameters are not valid for the whole composition as they are influenced by the individual routing and processing of a component. By convention, we use lower case variables whenever a parameter or model belongs to a component. In a composition, a component is identified by the subscript x which is a placeholder for a component abbreviation such as LB, S or W.

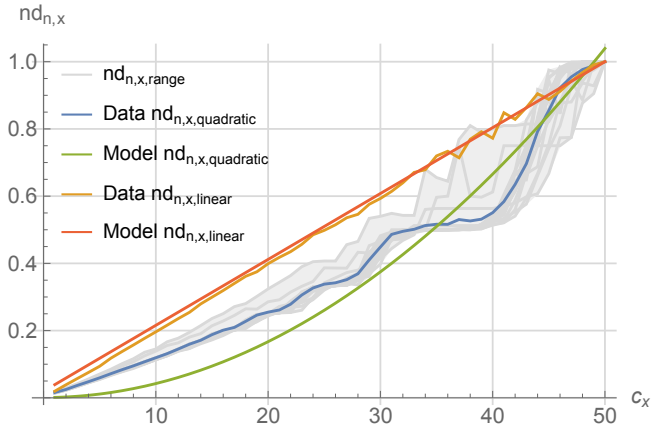

Fig. 3. Normalised measurments and model of the linear and quadratic network delay

5.2.2 Delay Factors

A request flowing through a component x is delayed by different factors. The model takes this into account by calculating the processing-, request size- and network delays:

$$d_{P,x} = d_{p,x} + d_{pp,x} \tag{1}$$

$$d_{S,x} = d_{s,x} \cdot s_x \tag{2}$$

For the network delay we observe two different developments from measured data that is illustrated in Fig. 3:

$$d_{N,x} = \begin{cases} \dfrac{c_x \cdot d_{g,x}}{m_x} + d_{n,x}, & \text{if } \textit{linear} \quad (3a) \\ \dfrac{c_x^2 \cdot d_{g,x} + d_{n,x}}{m_x}, & \text{if } \textit{quadratic} \quad (3b) \end{cases}$$

The linear data (Fig. 3) is retrieved from a cache C with constant lookup time, where the quadratic data comes from the measurements of an application A. Despite the increased complexity, we offer both models and allow the user to select the appropriate accuracy. If simplicity is chosen over accuracy, the linear version can be used only. Otherwise, the delay model is selected by the model fit.

5.2.3 Maximum Request Flow

To calculate the requests that can flow through a component per second, the concurrency is divided by the sum of all component delay factors. Adding more machines to the component increases the flow f_x by m_x to a maximum:

$$f_x = \frac{c_x \cdot m_x}{d_{P,x} + d_{S,x} + d_{N,x}} \tag{4}$$

As the performance improvement is not linear with m_x, $d_{N,x}$ increases with m_x and thereby degrades the performance.

TABLE 5
Composition parameters that are used to describe and model the performance of the composition of multiple components.

Goal-oriented	**Description**
Request-Flow / Second: $F \in [0, \infty]$	The requests that flow through all components in one second.
Target Flow / Second: $T \in [0, \infty]$	A desired target F for the whole system.
Workload-based	**Description**
Cache-Processing Ratio: $CPR \in [1, 0]$	The relation between the read-requests R_R and the processing-requests R_P.
Cache-Hit Ratio (V_N): $CHR \in [0, 1]$	The relation between cache-hits and cache-misses.
Deployment-based	**Description**
Machines: $M \in [1, \infty]$	The number of total machines that are used for the whole system.
Machine-Quantity Tuple: $MQT = (m_x \mid x \in C_X)$	Lists the number of machines for each component in a composition X.

5.2.4 Machines for Target Flow

To satisfy a target flow $f_x = t_x$ the number of machines a component uses needs to be adapted. It can be calculated by solving f_x for m_x in the linear case l and the quadratic case q (a and b represent substitutions for display purpose only):

$$\begin{aligned} a_l &= d_{p,x}t_x + d_{pp,x}t_x + d_{s,x}s_x t_x \\ b_l &= \sqrt{t_x(4c_x^3 d_{g,x} + 4c_x d_{n,x} + (d_{p,x} + d_{pp,x} + d_{s,x}s_x)^2 t_x)} \\ a_q &= d_{n,x}t_x + d_{p,x}t_x + d_{pp,x}t_x + d_{s,x}s_x t_x \\ b_q &= \sqrt{t_x(4c_x^2 d_{g,x} + (d_{n,x} + d_{p,x} + d_{pp,x} + d_{s,x}s_x)^2 t_x)} \\ m_{t,x} &= \left\lceil \frac{1}{2c_x}(a_l + b_l) \right\rceil, \quad \text{or} \quad \left\lceil \frac{1}{2c_x}(a_q + b_q) \right\rceil \end{aligned} \tag{5}$$

For the normal version $V_N : d_{pp,x} = 0$, as the normal version has no concept of post-processing.

5.3 Composition Models

The composition models compose the individual components into a larger system.

5.3.1 Parameters

The parameters denoted in Table 5 are valid for a whole composed system of components. By convention, we use capital notation whenever a parameter or model belongs to the whole composition.

5.3.2 Components and Sub-Systems

We propose two compositions of components: The normal version V_N and the scaled version V_S. V_N uses the components $C_N = (LB, A, C)$ and $V_S : C_S = (LB, S, C, Q, W, PS)$. For the model the components need to be separated by read sub-system SR and processing sub-system SP:

$$C_{N_{SR,SP}} = (LB, A) \tag{6}$$
$$C_{N_{SR}} = (C) \tag{7}$$
$$C_{S_{SR,SP}} = (LB, S) \tag{8}$$
$$C_{S_{SR}} = (C) \tag{9}$$
$$C_{S_{SP}} = (Q, W, PS) \tag{10}$$

5.3.3 Maximum Request Flow

The maximum request-flow F predicts the maximal throughput of V_N or V_S for a machine-quantity tuple MQT. For V_N, requests can be either served (*hit*) or need to be processed (*miss*) depending on the cache C's cache-hit ratio CHR:

$$d_{p,A} = (CPR \cdot d_{l,A}) + (d_{p,A} - CPR \cdot CHR \cdot d_{p,A}) \tag{11}$$

The concurrency at the cache C depends on the number of machines and concurrency of the application A and the cache-processing ratio CPR:

$$c_C = m_A \cdot c_A \cdot CPR \tag{12}$$

The maximum flow is determined by the slowest component of a composition:

$$F_N = \min\{f_x \mid x \in C_{N_{SR,SP}}, \frac{f_x}{CPR} \mid x \in C_{N_{SR}}\} \tag{13}$$

For the scaled version V_S, both the read sub-system SR and the processing sub-system SP have to be considered:

$$\begin{aligned} F_S = \min\{&f_x \mid x \in C_{S_{SR,SP}}, \frac{f_x}{CPR} \mid x \in C_{S_{SR}}, \\ &- \frac{f_x}{-1 + CPR} \mid x \in C_{S_{SP}}\} \end{aligned} \tag{14}$$

5.3.4 Machines for Target Flow

The machines equation calculates the machine-quantity tuple MQT for a certain target flow $F = T$ in a composition. For V_N, the app delay $d_{p,A}$ is as shown in (11) and the cache C is only hit by read-requests R_R:

$$t_C = CPR \cdot T \tag{15}$$

The quantities are calculated for every component:

$$M_{T,N,MQT} = (m_x \mid x \in C_N) \tag{16}$$
$$= (m_{LB}, m_A, m_C) \tag{17}$$
$$M_{T,N} = \sum_{m_x, x \in C_N} m_x \tag{18}$$

The tuple (17) is turned to a scalar by summing its individual components (18). The sum $M_{T,N}$ is the total number of machines needed for target T. For instance, if the load-balancer uses three machines, the application six machines and the cache a single machine, the tuple looks like this:

$$M_{T,N,MQT} = (3, 6, 1) \tag{19}$$
$$M_{T,N} = 10 \tag{20}$$

The number of total machines for the target T then is the sum of all component machines, in this example 10.

For V_S, the target T is split up by the processing sub-system SP and the read sub-system SR:

$$t_x = \begin{cases} T, & \text{if } x \in C_{SR,SP} \quad (21a) \\ CPR \cdot T, & \text{if } x \in C_{SR} \quad (21b) \\ (1 - CPR) \cdot T, & \text{if } x \in C_{SP} \quad (21c) \end{cases}$$

The MQT is generated from all components:

$$M_{T,S,MQT} = (m_x \mid x \in C_S) \tag{22}$$
$$= (m_{LB}, m_S, m_C, m_Q, m_W, m_{PS}) \tag{23}$$
$$M_{T,S} = \sum_{m_x, x \in C_S} m_x \tag{24}$$

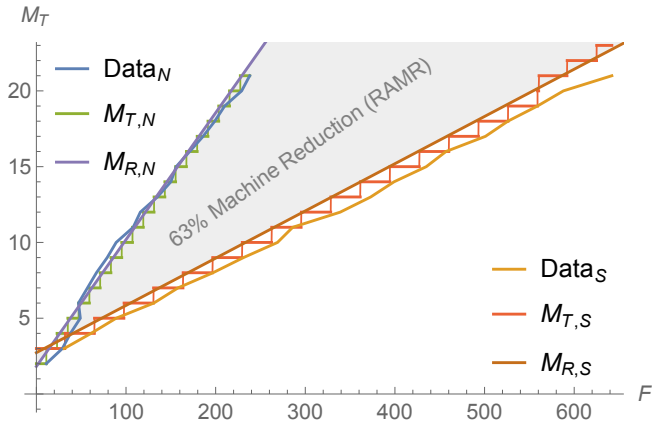

Fig. 4. A comparison of the total machines for target model M_T, the linear total machines regression M_R and measured data for the scaled version V_S and normal version V_N. The difference between V_S and V_N is the relative average machine reduction $RAMR$.

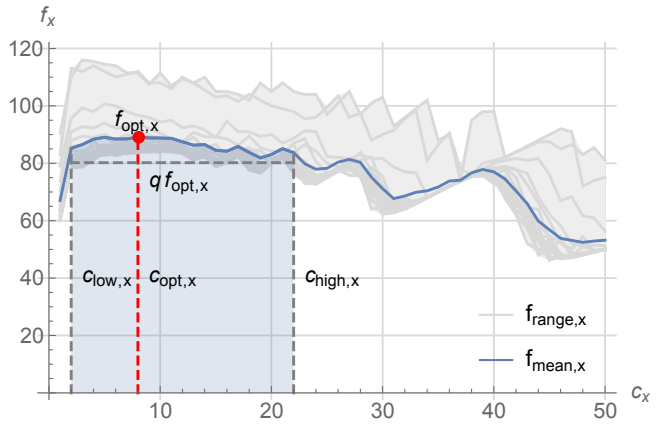

Fig. 5. Optimal Concurrency Range $ocw_{0.9,x} = [2, 22]$ with a performance-concurrency-width triplet $pcw_{0.9,x} = (89, 8, 20)$ for the average normalised request-flow f_x of 20 machines.

As for the normal version V_N, the tuple (23) is turned to a scalar by summing its individual components (24).

5.3.5 Linear Machines Regression for Target Flow

The linear regression for the number of machines provides a simpler approximation of the performance (Fig. 4). For the normal version V_N, the app delay $d_{p,A}$ is as shown in (11). The slope can be calculated as a unit of increasing total machines per flow-quantity:

$$M_{R,N,s} = \sum_{x \in C_{N_{SR,SP}}} \frac{1}{f_x} + \sum_{x \in C_{N_{SR}}} \frac{CPR}{f_x} \tag{25}$$

The full regression equation multiplies the slope with the target T and adds the minimal number of component machines $|C_N|$:

$$M_{R,N} = T \cdot M_{R,N,s} + |C_N| \tag{26}$$

For the scaled version V_S, the slope needs to consider both the read sub-system SR and the processing sub-system SP:

$$\begin{aligned} M_{R,S,s} = &\sum_{x \in C_{S_{SR,SP}}} \frac{1}{f_x} + \sum_{x \in C_{S_{SR}}} \frac{1}{CPR \cdot f_x} \\ &+ \sum_{x \in C_{S_{SP}}} \frac{1 - CPR}{f_x} \end{aligned}$$

$$M_{R,S} = T \cdot M_{R,S,s} + |C_S| \tag{27}$$

5.4 Performance Comparison

To evaluate the performance of our proposed compositions V_N and V_S, we develop comparison metrics.

5.4.1 Relative Average Machine Reduction

When the scaled version V_S needs fewer machines for the same load than the normal version V_N, we express the delta as a factor of machine reduction. The relative average machine reduction is calculated with the slopes of the linear total machines regressions of both versions (Fig. 4).

$$RAMR = 1 - \frac{M_{R,S,s}}{M_{R,N,s}} \tag{28}$$

If the scaled version V_S needs five machines and the normal version V_N needs six machines for the same load, the RAMR equals $(1-(5/6)) = 0.17$. This shows, that the scaled version V_S needs 17% fewer machines than the normal version V_N. If both versions use the same number of machines, the $RAMR$ is zero.

5.4.2 Break-Even Point for Post-Processing

In the normal version V_N, the app A is responsible for the cache updates and invalidation. Therefore, the cost of updates is added to to the app delay $d_{p,A}$. In the scaled version V_S, the post-processing delay $d_{pp,W}$ is explicitly defined as the time it takes to post-process a request. When comparing both versions, an interesting metric is the time the scaled version V_S has available for the post-processing $d_{pp,W}$. The post-processing delay $d_{pp,W}$ where both versions deliver the same performance is the break-even point $d_{pp,W,BEP}$. It can be calculated by equalising the linear regressions of the normal version V_N and scaled version V_S and solved for the post-processing delay $d_{pp,W}$:

$$C_{S_{SP}} = C_{S_{SP}} \setminus \{W\} \tag{29}$$

$$\begin{aligned} d_{pp,W,BEP} &= (M_{R,N} = M_{R,S}), \quad \text{solve for } d_{pp,W} \\ &= c_w \cdot m_W \cdot (M_{R,N} - M_{R,S}) \\ &\quad - (1 - CPR) \cdot (d_{N,W} + d_{S,W} + d_{p,W}) \end{aligned} \tag{30}$$

For the break-even calculation, the worker component W is excluded from the processing components $C_{S_{SP}}$ as shown in (29). The worker component delays without the post-processing delay are specifically considered in the last term of (30).

5.5 Performance Optimisation

To optimise the performance of a component we consider the optimal load and implementation specific metrics.

5.5.1 Optimal Concurrency Range

From our machine-normalised measurements for one to 22 machines as $f_{range,x}$ (Fig. 5) we observed the request flow of a component to have an optimal range. $f_{mean,x}$ shows that with increasing concurrency c_x, a component has a

performance optimum $c_{opt,x}$ where it delivers the maximal request flow $f_{opt,x}$. An algorithm that controls the flow of requests to a component will try to load the component with the optimal concurrency $c_{opt,x}$. A key metric for a component, however, is the sensitivity around the optimum concurrency. If the performance degradation is low around the optimum, algorithms can operate with higher tolerance. This allows the systems to be more insensitive to dynamic concurrency values, which leads to fewer scaling actions. To describe this broadness of possible concurrency values with respect to a percentage request flow loss q, we introduce the optimal concurrency range $ocr_{q,x}$. An $ocr_{0.95,x} = [5, 20]$ means that a component is able to handle concurrencies between 5 and 20 while operating at 95% of the performance optimum $f_{opt,x}$.

The optimal concurrency range $ocr_{q,x}$ can be calculated from a series of performance $data(c_x)$ that is measured with increasing concurrency values c_x. The following is representative pseudocode for finding the lowest- and highest concurrency values:

1: $f_{opt,x} \leftarrow \max(data)$
2: $c_{low,x} \leftarrow c_{high,x} \leftarrow c_{opt,x} \leftarrow \max^{-1}(f_{opt,x})$
3: **while** data$(c_{low,x}) \geq q \cdot f_{opt,x}$ **do**
4: $\quad c_{low,x} \leftarrow c_{low,x} - 1$
5: **end while**
6: **while** data$(c_{high,x}) \geq q \cdot f_{opt,x}$ **do**
7: $\quad c_{high,x} \leftarrow c_{high,x} + 1$
8: **end while**
9: **return** $[c_{low,x}, c_{high,x}]$

The optimal concurrency width $ocw_{q,x}$ of an optimal concurrency range $ocr_{q,x}$ expresses the general sensitivity of the component to concurrency:

$$ocw_{q,x} := \max(ocr_{q,x}) - \min(ocr_{q,x}) \tag{31}$$

5.5.2 Performance-Concurrency-Width Triplet

In order to optimise the performance, we propose the Performance-Concurrency-Width triplet $pcq_{q,x}$ as a metric that allows comparing different implementations of components with each other.

$$pcw_{q,x} = (f_{opt,x}, c_{opt,x}, ocw_{q,x}) \tag{32}$$

The $pcw_{q,x}$-triplet includes the most important performance parameters for a component. It allows building a performance delta triplet $\Delta pcw_{q,x} = pcw_{q,Z} - pcw_{q,Y}$ that shows the performance differences between implementation Y and Z. A delta triplet $\Delta pcw_{q,x} = (12, 0, 8)$ presents Z as a superior implementation to Y. With the same concurrency, implementation Z's request-flow f is 12 requests bigger. At the same time it is 8 concurrency values more insensitive to load.

6 Empirical Prototype Evaluation

We empirically evaluate the proposed prototype with multiple machines on the component and composition level. Additionally, we compare the performance of three real-world applications for the previously defined normal version V_N and scaled version V_S compositions.

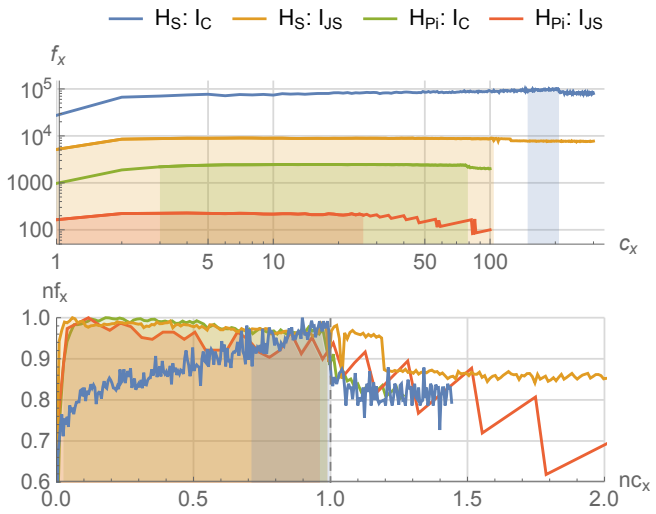

Fig. 6. Absolute and normalised performance comparison of a JavaScript-implementation I_{JS} and a C-implementation I_C on server hardware H_S and Pi-computers H_{Pi}.

6.1 Relation to our Previous Work

In [1] we evaluated a model that considered multiple machines in the single-machine scope only, where in this work we evaluate the proposed WSF prototype with multiple machines. The normal version V_N in [1] did not employ a cache, where this work enables caching for V_N for a fairer comparison. Additionally, in this work we evaluate the performance using real-world application traces where our previous work used a defined set of common parameter ranges in simulated network traffic.

6.2 Evaluation System

Before we evaluate our model, we test our evaluation system for the influences of different programming frameworks and hardware.

6.2.1 Implementation

In general, the implementation for our evaluation system consists of three modules: A *test-runner* that coordinates the cluster execution and collection of results, a *traffic-generator* that generates the load and sends it to selected machines, and a *component-machine* that either responds to a request or passes it on to another component-machine. All modules are configurable with different parameters that control delay, size of requests, concurrency and machine selection.

6.2.2 Influence of Programming Framework

To compare the influence of the programming framework on the results we create two implementations: A JavaScript-implementation I_{JS} on *node.js* [25] and a C-implementation I_C using *libuv* [26]. We notice the C-implementation I_C is 1 order of magnitude faster than the JavaScript-implementation I_{JS} (Fig. 6).

6.2.3 Influence of Hardware

To evaluate the effect of different hardware, both implementations are tested on: Server hardware H_S using a 2.6

TABLE 6
Isolated evaluation data for the delay factors of the component models.

Delay	Param	Value	R^2	RMSE	Fit
Network	$d_{n,x}$	8.32×10^{-4}	0.975	1.7×10^{-2}	0.973
	$d_{g,x}$	3.96×10^{-4}			
Size	$d_{s,x}$	1.26×10^{-4}	0.998	1.3×10^{-3}	0.975
Process	$d_{p,x}$	2.07×10^{-10}	—	—	—

GHz Intel Core i7 and a Raspberry Pi computer H_{Pi} with 700 MHz ARMv6l single-core where both are running Arch Linux. We notice the server hardware H_S is 1 order of magnitude faster than the Pi-computers H_{Pi}.

6.2.4 Results

The data supports that the results of the evaluation are transferable between different implementations and hardware. Fig. 6 illustrates the normalised and absolute performance curves of both implementations on both hardware. All normalised curves show the same performance pattern where the performance slowly increases to a maximum and then breaks down as the system is overloaded. The performance of the C-implementation I_C on the server hardware H_S increases notably slower than all other implementations. This is due to the enormous concurrency that is needed to reach the maximum performance. This slower growth is reflected in the Performance-Concurrency-Width triplet pcw_q, where the implementation needs high concurrency values to run close to its optimal performance.

As the usability of the JavaScript-implementation I_{JS} is better, it is selected for the evaluation of the prototype. The implementation is deployed in a cluster of 42 Raspberry Pi Model B single-board computers H_{Pi} that are connected through a HP ProCurve 2810–48G switch using 100 Mbit/s ethernet.

6.3 Component Models Evaluation

Firstly, we evaluate the component models using a single component with multiple machines. The central equation, the maximum request flow f_x of a component is composed of the number of machines m_x, the network delay $d_{N,x}$, the request size delay $d_{S,x}$ and the process delay $d_{P,x}$. Since the evaluation of the whole equation is complicated, the delay parts are isolated and evaluated individually where the results are given in Table 6.

6.3.1 Metrics

To quantify the results of the evaluation and compare the model to the data, we calculate the Coefficient of Determination (R^2), Root-Mean-Square Error (RMSE), Normalised RMSE (NRMSE) and model fit. The RMSE shows the absolute error without relating it to the range of observed values. The normalised version of the RMSE relates to the observed values so that $NRMSE = RMSE/(y_{\max} - y\min)$ where $y_{\max}$ and $y_{\min}$ represent the maximum and minimum of all observed values y. This allows expressing the model fit $Fit = 1 - NRMSE$ as a percentage where 1.0 is a perfect fit and 0.0 is no fit.

6.3.2 Network Delay

In the first step, the network delay $d_{N,x}$ is isolated by setting $d_{S,x} = d_{P,x} = 0$. For the model, the known parameters are the number of machines m_x and the concurrency c_x. The tests are run for all possible (m_x, c_x) combinations where the number of machines m_x is in the range of $(1 \dots 20)$, and the number of concurrent requests c_x is in the range of $(1 \dots 50)$. The network delay $d_{n,x}$ can be retrieved from the results as the smallest measured delay. The network delay gain $d_{g,x}$ can either be formulated as a quadratic- or linear optimisation problem on the network delay $d_{N,x}$. It is solved using Mathematica's NonlinearModelFit which automatically picks a linear or quadratic delay that fits best to the regression methods listed in [27]. The results support the network delay model $d_{N,x}$ as it fits the data by 97.3%.

6.3.3 Request Size Delay

The size delay $d_{S,x}$ is isolated by setting the processing delay $d_{P,x} = 0$ and the concurrency and number of machines $c_x = m_x = 1$. According to the HTTP Archive [28], the average individual response size depends on the content-type but is smaller than 108 kB. Tested sizes s range from 0 to 400 to cover the average response sizes listed in [28] well. The determination of the size delay $d_{s,x}$ can be formulated as a linear optimization problem with $d_{S,x}$ and is computed using Mathematica [27]. The results in Table 6 support the size delay model $d_{S,x}$ as it fits the data by 97.5%.

6.3.4 Processing Delay

The processing delay $d_{P,x}$ can simply be measured where our results are listed in Table 6. For example a component that guarantees a constant lookup time $O(1)$ is expected to have a constant processing delay $d_{P,x}$. For the cache component, we use *Redis* [29].

6.4 Composition Models Evaluation

As a next step, the composition evaluation is used to analyse the interplay of components. We conceive the *chain* and *distributed composition* of components where our proposed model makes the following assumptions that can be formulated as hypotheses:

- H_1: *The maximum request-flow is determined by the slowest component in the chain (chain composition).*
- H_2: *The maximum request-flow is relative to the distribution of the traffic to the components (distributed composition).*

6.4.1 Chain Composition

In the chain composition multiple components are stringed through a single connection. The evaluation is run with $(2 \dots 10)$ machines stringed together. One random machine in the chain introduces a processing delay of $d_{P,x} = 0.05$. Measured at the end of the chain, the expected maximum delay is:

$$F = (d_{N,x} + d_{P,x})^{-1} = 19 \tag{33}$$

The measurements of all chains show a request flow F of 19 which supports H_1 with empirical data.

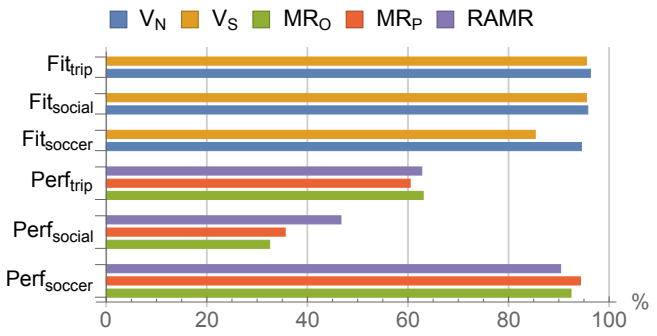

Fig. 7. Prediction fits Fit and machine reductions MR for all real-world applications in relation.

6.4.2 Distributed Composition

In the distributed composition one component distributes the requests to many others. One component X dispatches the traffic to two other components (Y, Z) with 10 different ratios $r_{disp} \in (0.0, 0.1, \ldots, 1.0)$. X introduces a processing delay of $d_{P,X} = 0.01$ and Y a processing delay of $d_{P,Y} = 0.1$. The expected maximum for each ratio is:

$$F = ((r_{disp} \cdot d_{P,X}) + ((1 - r_{disp}) \cdot d_{P,Y}))^{-1} \quad (34)$$

All test cases combined have a total $RMSE = 4.11$ and a total prediction fit of $Fit = 0.942$. This allows to support H_2 as the data supports the model with confidence of 94.2%.

6.5 Real-World Application Evaluation

After the evaluation of components and their composition, we evaluate the model as a whole. Therefore it is interesting to see how the model behaves when traffic with the parameters of three real-world application traces is applied.

For each application, both the normal version V_N and the scaled version V_S are implemented. We evaluate the collected data series of each application with both versions in two dimensions: The prediction fits of our proposed model and the machine reductions. The data series measure the achieved request-flow F for the total number of machines $M_{T,N}$ starting with the minimum determined by the components for the version $|C_N|$ and $|C_S|$ up to the maximum 20, as half of the 42 machines are needed for load generation. As a scaling decision, the new machine of the next run is always added to the slowest component that introduces the bottleneck.

6.5.1 Metrics

From each application trace we extract the parameters for the cache-processing ratio CPR, the processing-delay $d_{P,x}$ and the size s where the extracted parameters are listed in Table 7. We use the same prediction fit metrics as introduced in Section 6.3.1 to compare the data series with data calculated by our proposed model where the fits are given in Table 7. The Machine Reduction (MR) is used to compare the composite number of machines needed by the normal version V_N to the V_S with equal target request flows T. Table 7 shows the observed- and predicted machine reductions MR and the relative average machine reduction $RAMR$. In order to benefit from the optimised request flow, our proposed prototype in V_S needs to spend time to post-process requests. The normal version V_N spends the time to manage its cache implicitly in the measured application delay $d_{p,A}$. In order to determine the available time V_S has for post-processing while achieving equal performance to V_N, we calculate the post-processing delay break-even point $d_{pp,W,BEP}$ that is shown in the column PP BEP in Table 7.

6.5.2 Trip Planner

The trip planner T_{trip} is a web service that allows users to plan a journey all over the world. It calculates the itinerary between two or more destinations and enhances it with local information, e.g. restaurants and hotels. The service has no social features that allow the sharing of trips or recommendations. The traffic resembles an application of an intermediate update nature as trip indices can be calculated offline, but user input has to be handled. We are able to analyse $|T_{trip}| = 10$ million traces that are available at [30].

6.5.3 Social Network

The social network T_{social} is a platform we implemented and set up on campus as traffic traces of social-networks were unavailable. The platform provides a subset of the features of a typical social-network platform and is built to be as similar to Facebook as possible. In addition to the management of persons and their friendships, it shows a news feed with status messages from friends and allows exchanging private messages between friends. Missing features are photo- and video-sharing and the creation and management of groups. The platform is hosted on a university server and only accessible from the university test network. Over a time period of two months we recorded user requests and replayed them ten times against the platform. The traffic resembles traffic of a social nature as resources are constantly changed by users. This allowed us to collect a total of $|T_{social}| = 0.6$ million traces that are available at [30].

6.5.4 FIFA Soccer Worldcup 98 Website

Traces of the 1998 soccer World Cup website T_{soccer} between April 30, 1998 and July 26, 1998. The website resembles an application of a more static nature as no social features and few processing-requests are issued. We are able to analyse $|T_{soccer}| = 14$ million traces that are freely available at [31]. As T_{soccer} does not contain processing delays $d_{P,x}$, we manually set it to a rounded average processing delay $d_{P,x} = 1$ derived from TP_{trip} and TP_{social}.

6.6 Results

We evaluated our proposed model on three levels: The component level to ensure our proposed calculation with delay factors is feasible, the composition level to support our models regarding the interplay of components and the application level to verify our performance comparison models and compare the performance to a contemporary two-tier application pattern. We are able to support the component models as our results in Table 6 fit the data with sufficient accuracy. Both hypotheses H_1 and H_2 are supported by the composition models evaluation where the prediction fit for H_2 is 94.2%. Fig. 7 and Table 7 illustrate the results of the real-world application evaluation. The

TABLE 7
Trace Parameters, Prediction Fit, Machine Reduction and Post-Processing BEP for Real-World Applications

	Trace Parameters			Prediction Fit				Machine Reduction			PP BEP
App	CPR	$d_{P,x}$	s	$RMSE_N$	$RMSE_S$	Fit_N	Fit_S	Observed MR	Predicted MR	$RAMR$	$d_{pp,x}$
trip	0.849	1.41	2	0.707	0.858	0.962	0.954	0.630	0.604	0.627	2.69
social	0.571	0.48	161	0.806	0.858	0.957	0.954	0.325	0.355	0.466	1.19
soccer	0.998	1	5	1.048	2.790	0.944	0.853	0.924	0.943	0.903	26.04

average prediction fit of 93.7% for all three applications further supports our proposed model. All applications in Table 7 need fewer machines (63%, 32%, 92%) if they use the proposed prototype, which demonstrates the potential of our proposal. Because the normal version V_N routes all requests to the application component, the scaled version V_S has 2.69, 1.19 and 26.04 seconds to post-process requests before the performance of both versions is equal.

Finally, it is noted that the authors are the ones who evaluated their own implementation with a potential bias, where in the future independent instances should verify the results. Furthermore, it would be interesting to see that the proposed model is validated with further extensive, more contemporary datasets beyond the traffic traces of the 1998 soccer World Cup website.

7 Conclusion and Future Work

The proposed WSF builds upon existing web service infrastructure to be directly applicable to today's web services, and a novel scaling layer is integrated to enable scaling capability for existing web services in the cloud. In this work, we developed a conceptual architecture of WSFs with components, modules, interfaces and a set of optimised schemes for scaling including request flow routing, caching, processing, performance profiling etc. We extended the mathematical model of the prototype we proposed in our previous work. A cluster of 42 Raspberry Pis allowed us to successfully evaluate the model in the multi-machine cloud scope. We were able to put the performance evaluation into a real context by using the parameters extracted from a total of 25 million trip planner, social network and soccer worldcup traces. We introduced a cache to the traditionally-scaled version to provide a fairer comparison with the WSF version. We also presented the development-process for a web platform when a WSF is applied.

The results showed that the application of a WSF can reduce the number of total machines needed by 32%, 63% and 92%. When the traditionally-scaled version does not spend any time on cache-updates, this allows the WSF to occupy 2.7s, 1.2s and 26s for synchronous post-processing updates. If a WSF is able to process all request dependencies within these time limits, it outperforms traditionally-scaled WAF systems.

The focus of our future work will be on further optimisation of the post-processing system. We will conduct research in the area of dependency analysis to find suitable data structures for dependencies that allow a practical declaration, management and analysis of resource links. Furthermore, we will search for eligible, parallelisable algorithms that optimise the post-processing performance. Optimisations will include mechanisms such as the detection and removal of duplicate updates, the breakup of cyclic updates and the introduction of update rate limits for resources. We will also use developer assistive systems to point out critical deep dependency paths to enable developers the decoupling of affected resources.

References

[1] T. Fankhauser, Q. Wang, A. Gerlicher, C. Grecos, and X. Wang, "Web scaling frameworks: A novel class of frameworks for scalable web services in cloud environments," in *Proc. IEEE Int. Conf. on Commun. (ICC14)*, June 2014, pp. 1414–1418.

[2] C. Krintz, "The appscale cloud platform: Enabling portable, scalable web application deployment," *J. IEEE Int. Comp.*, vol. 17, no. 2, pp. 72–75, March 2013.

[3] A. Wolke and G. Meixner, "Twospot: A cloud platform for scaling out web applications dynamically," in *Towards a Service-Based Internet*, E. Di Nitto and R. Yahyapour, Eds. Springer Berlin Heidelberg, 2010, vol. 6481, pp. 13–24.

[4] R. Han, M. M. Ghanem, L. Guo, Y. Guo, and M. Osmond, "Enabling cost-aware and adaptive elasticity of multi-tier cloud applications," *Future Gener. Comput. Syst.*, vol. 32, pp. 82–98, 2014.

[5] J. Espadas, A. Molina, G. Jimnez, M. Molina, R. Ramrez, and D. Concha, "A tenant-based resource allocation model for scaling software-as-a-service applications over cloud computing infrastructures," *Future Gener. Comput. Syst.*, vol. 29, no. 1, pp. 273–286, 2013.

[6] J. Jiang, J. Lu, G. Zhang, and G. Long, "Optimal cloud resource auto-scaling for web applications," in *Proc. IEEE/ACM Int. Sym. Cluster, Cloud and Grid Comp. (CCGrid13)*, May 2013, pp. 58–65.

[7] A. Gambi, W. Hummer, H.-L. Truong, and S. Dustdar, "Testing elastic computing systems," *J. IEEE Int. Comp.*, vol. 17, no. 6, pp. 76–82, Nov 2013.

[8] W. Hummer, B. Satzger, and S. Dustdar, "Elastic stream processing in the cloud," *Wiley Interdis. Rev.: Data Mining and Knowl. Disc.*, vol. 3, no. 5, pp. 333–345, 2013.

[9] N. Le Scouarnec, C. Neumann, and G. Straub, "Cache policies for cloud-based systems: To keep or not to keep," in *Proc. IEEE Int. Conf. on Cloud Comp. (CLOUD14)*, June 2014, pp. 1–8.

[10] X. Qin, W. Zhang, W. Wang, J. Wei, H. Zhong, and T. Huang, "Online cache strategy reconfiguration for elastic caching platform: A machine learning approach," in *Proc. IEEE Ann. Comp. Soft. and App. Conf. (COMPSAC11)*, July 2011, pp. 523–534.

[11] E. Bocchi, M. Mellia, and S. Sarni, "Cloud storage service benchmarking: Methodologies and experimentations," in *Proc. IEEE Int. Conf. on Cloud Net. (CloudNet14)*, Oct 2014, pp. 395–400.

[12] H. Han, Y. C. Lee, W. Shin, H. Jung, H. Yeom, and A. Zomaya, "Cashing in on the cache in the cloud," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 8, pp. 1387–1399, Aug 2012.

[13] R. Pettersen, S. Valvag, A. Kvalnes, and D. Johansen, "Jovaku: Globally distributed caching for cloud database services using dns," in *Proc. IEEE Int. Conf. on Mob. Cloud Comp. (MobileCloud14)*, April 2014, pp. 127–135.

[14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter, "Composite cloud application patterns," in *Cloud Computing Patterns*. Springer Vienna, 2014.

[15] (2014) Microservices. Martin Fowler. [Online]. Available: http://martinfowler.com/articles/microservices.html

[16] D. Namiot and M. Sneps-Sneppe, "On micro-services architecture," *Int. J. Open Inf. Technol.*, vol. 2, no. 9, pp. 24–27, 2014.

[17] (2014) Cqrs. Martin Fowler. [Online]. Available: http://martinfowler.com/bliki/CQRS.html

[18] (2005) Ibm an architectural blueprint for autonomic computing. [Online]. Available: http://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf
[19] C. Fehling, F. Leymann, and R. Retter, "Your coffee shop uses cloud computing," *J. IEEE Int. Comp.*, vol. 18, no. 5, pp. 52–59, Sept 2014.
[20] (2010) Cqrs documents. Greg Young. [Online]. Available: https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf
[21] (2014) Oasis topology and orchestration specification for cloud applications version 1.0. [Online]. Available: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html
[22] (2014) Docker. [Online]. Available: http://docker.com
[23] (2014) Amazon Web Services. [Online]. Available: http://aws.amazon.com
[24] (2014) Google. [Online]. Available: https://cloud.google.com
[25] (2014) Joyent, Inc. [Online]. Available: http://nodejs.org
[26] (2014) Joyent, Inc. [Online]. Available: https://github.com/joyent/libuv
[27] (2014) Wolfram. [Online]. Available: http://reference.wolfram.com/language/ref/NonlinearModelFit.html
[28] (2014) Internet Archive, a 501(c)(3) non-profit. [Online]. Available: http://httparchive.org
[29] (2014) Open Source. [Online]. Available: http://redis.io
[30] (2015) Web Scaling Frameworks Traces. [Online]. Available: http://webscalingframeworks.org/traces
[31] (1998) Hewlett-Packard Company. [Online]. Available: http://ita.ee.lbl.gov/html/contrib/WorldCup.html

Thomas Fankhauser (SM13) received his Bachelor and Master degree in computer science from Stuttgart Media University, Germany. He is currently pursuing a PhD degree at the University of the West of Scotland while working as an academic research assistant at Stuttgart Media University. His research interests include the scalability of web services, software architectures and frameworks in cloud computing environments. Since 2013 he is a student member of the IEEE Consumer Electronics Society. He was a Best Poster Award Winner of UWS ICTAC 2013 and has published at the IEEE flagship conference ICC 2014 in Sydney and is author of a book on social phenomena in social networking services. Previously, he worked as a software architect and developer in social web and mobile advertising industries for several years.

Qi Wang (S02-M06) is a Senior Lecturer in Computer Networks with the University of the West of Scotland (UWS), UK. He is the Principal Investigator (PI) of the UK EPSRC project Enabler for Next-Generation Mobile Video Applications (EP/J014729/1), and Co-PI and Co-Technical Manager for the EU Horizon 2020 5G-PPP project SELFNET: Framework for Self-Organised Network Management in Virtualized and Software Defined Networks (H2020-ICT-2014-2/671672). His research interests include video networking and processing, mobile/wireless networks, cloud computing, and network management. He has published over 70 papers in these areas. He was a Best Paper Award Winner of IEEE ICCE 2014, IEEE ICCE 2012 and SIGMAP 2014, and Best Paper Award Finalist of IEE 3G2003. He received his PhD degree in Mobile Networking from the University of Plymouth, UK.

Ansgar Gerlicher (M06) is a Professor in Mobile Applications and Director of Research in Mobile Applications & Security with the Institute of Applied Science, Stuttgart Media University, Germany. Previously, he worked for several years as a Software Architect and Project Manager in the Telecommunication and Automotive Industries. He is on the programme committee of the Apps To Automotive Conference, Stuttgart and published several papers in international conference proceedings and in the Springer Lecture Notes in Computer Science. He is co-author of several books on mobile software development and computer science and media in Germany. His research interests include integration of consumer electronic devices in vehicles and mobile and embedded software architectures, frameworks and mobile security. He received his PhD degree in real-time collaboration systems from the London College of Communication, UArts, UK. Since 2012 he is a member of the IEEE Consumer Electronics Society.

Christos Grecos (SM IEEE 06, SM SPIE 2008) is an Independent Imaging Consultant at Glasgow, UK. He worked as a Professor in Visual Communications Standards, and Head of School of Computing in the University of the West of Scotland and previously in the Universities of Central Lancashire and Loughborough, all in UK. His research interests include image/video compression standards, image/video processing and analysis, image/video networking and computer vision. He has published over 150 research papers in top-tier international publications including a number of IEEE transactions on these topics. He is on the editorial board or served as guest editor for many international journals, and he has been invited to give talks in various international conferences. He has obtained significant funding for his research as the Principal Investigator for several national or international projects funded by UK EPSRC or EU. He received his PhD degree in Image/Video Coding Algorithms from the University of Glamorgan, UK.

Xinheng Wang received his first degree and Master degree in electrical engineering from Xi'an Jiaotong University, China, in 1991 and 1994, respectively and Ph.D. degree from Brunel University in 2001. He is currently a professor in networks at University of the West of Scotland with research interests in wireless mesh networks, wireless sensor networks, Internet of Things, converged indoor location, and service-oriented networks. He has authored/coauthered more than 110 journal and conference papers and filed 7 patents. He is actively engaged with industry and acting as a member of BSI (British Standards Institution) Committee in ICT, Electronics and Healthcare. In addition, he has severed as an External Expert Adviser for the National Assembly for Wales since Nov. 2009.