Web Scaling Frameworks: A novel class of frameworks for scalable web services in cloud environments

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

***Abstract*—The social web and huge growth of mobile smart devices dramatically increases the performance requirements for web services. State-of-the-art Web Application Frameworks (WAFs) do not offer complete scaling concepts with automatic resource-provisioning, elastic caching or guaranteed maximum response times. These functionalities, however, are supported by cloud computing and needed to scale an application to its demands. Components like proxies, load-balancers, distributed caches, queuing and messaging systems have been around for a long time and in each field relevant research exists. Nevertheless, to create a scalable web service it is seldom enough to deploy only one component. In this work we propose to combine those complementary components to a predictable, composed system. The proposed solution introduces a novel class of web frameworks called Web Scaling Frameworks (WSFs) that take over the scaling. The proposed mathematical model allows a universally applicable prediction of performance in the single-machine- and multi-machine scope. A prototypical implementation is created to empirically validate the mathematical model and demonstrates both the feasibility and increase of performance of a WSF. The results show that the application of a WSF can triple the requests handling capability of a single machine and additionally reduce the number of total machines by 44%.**

I. Introduction

The enormous growth of smart mobile devices in combination with social web services increases the number of requests that need to be processed by modern web platforms in a timely fashion. Whereas cloud computing provides the ability to provision the hardware needed, state-of-the-art Web Application Frameworks (WAFs) do not offer integrated scaling concepts to deal with automatic resource-provisioning and elastic caching or ensure a guaranteed maximum response time.

They are rather designed to abstract common functionalities needed for web application development including data-management, url-mapping, session-handling and response-generation. Today, users progressively access the social web from anywhere using their mobile smart devices, which leads to increased traffic. A single computing resource might not be able to satisfy such an amount of requests - only the junction of multiple computing resources, where each resource gets a small share of the total requests, allows to handle such huge amounts of requests in aggregation. Handling the exponentially increasing global requests adds the requirement of being able to run multiple instances of an application for highly scalable web services. The major challenges that are introduced by this requirement are the management of the shared resources, the balancing of the requests among all instances and the decision when to spawn or terminate instances. These challenges are collectively referred to as horizontal scaling [13], [14], [16].

[*]School of Computing, University of the West of Scotland, Email: {Thomas.Fankhauser, Qi.Wang, Christos.Grecos, Xinheng.Wang}@uws.ac.uk
[†]Mobile Application Development, Stuttgart Media University, Email: {fankhauser, gerlicher}@hdm-stuttgart.de

Our experiments have showed that WAFs have different strengths and weaknesses. A highly abstracted WAF like Ruby on Rails, for example, was slower than the very thin WAF node.js but more powerful regarding data management and interface rendering. If a web service needs to provide both a fast and slim JSON API and a full blown HTML website it is the best solution to combine both WAFs. As both the horizontal scaling and web service composition are very complex matters, it makes sense not to introduce them to WAFs but offload them to another layer - the Web Scaling Framework (WSF) proposed in this paper. Fig. 1 illustrates a WSF that incorporates multiple WAF applications.

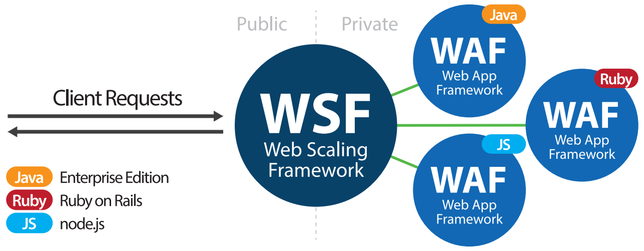

Fig. 1. **The relationship between the WSF and WAFs**

To comply to a proposed class of WSFs, a WSF should:

- take over the responsibilities of scaling and incorporate existing WAFs
- separate the business logic in the web service from the hosting logic
- connect to and combine existing WAFs to a compound web service using standard HTTP requests
- introduce low overhead when added, whilst adding the instant ability to scale
- constantly adapt their infrastructure to fit the required performance at all times

- be able to provision as much resources on a pay-per-use base as needed
- benefit from the horizontal scaling ability that is enabled by cloud computing
- be able to transparently use Software-as-a-Service or machine-cluster components

The paper is organised as follows. Section II summarises the results of the related work. Section III describes the overall design of the proposed WSF. Section IV introduces the modelling and prototyping of the proposed WSF in a single- and a multi-machine cloud scope, respectively. Section V presents the empirical results to evaluate the machine scope model and the analytical effect of adding more machines to the system. Section VI visualises the application of the proposed WSF by looking into three different web service scenarios that apply the proposed WSF. Section VII outlines the conclusions and gives a perspective on future research.

II. Related Work

For each of the components of the proposed WSF there exists a branch of research. Our focus, however, is on the composition of the components. We reviewed three perspectives *Architecture*, *Cache*, *Data Store* with seven attributes *Full Architecture*, *Algorithm*, *Testbed*, *Data*, *Queue*, *Load-Generation*, *Survey* in the existing work. We found that most publications focus on one perspective only. The TwoSpot platform [9] applies open and standards-based technologies and therefore composes reverse-proxies, work-servers, file-servers and load-balancers. The author's major focus is on preventing vendor lock-in by an open Platform-as-a-Service. ElasticSite [11] extends non-cloud resources by cloud resources with a focus on different launch policies. AzureBlast [10] combines components of the WindowsAzure Platform-as-a-Service to run the BLAST algorithm in parallel. Reference [12] proposes a highly resilient systems architecture for cloud with a focus on failures and fallback handling. Auto-Scaling [8] proposes an algorithm with job deadlines to optimise resource utilisation. There is reputable research for the different component branches. Join-Idle-Queue [6] is an improved novel class of queueing algorithms. The EQS [7] introduces a decent way to scales message queues and publish-subscribe systems in the cloud. An elastic cache system is introduced by [4] and [5] proposes a cache framework that performs comparable to sole remote accessed in-memory cache systems. The novelty of our work is the composition of existing components to form a scalable, predictable system. The performance of the system can be calculated with a mathematical model, both in the machine- and cloud scope. The model is based on component delays and can be easily extended to accommodate more components. The system uses the technology of existing and the future development of common WAFs and relieves WAF from scaling. It uses a persistent cache that is kept updated at all times to speed-up read-only responses. The proposed work is able to utilise both Infrastructure-as-a-Service and Software-as-as-Service as components.

TABLE I
Components of the proposed Web Scaling Framework

Cluster Component	**Description**
Servers (S)	Classify the request to be either $R_R : m \in$ {*GET*} or $R_P :\in$ {*POST, PUT, DELETE, ...*} by the HTTP method m. They lookup content from the cache system or enqueue requests to have them processed.
Caches (C)	Store the whole content the WSF is able to serve.
Queues (Q)	Hold the requests that need processing until they are popped by available Ws. Different priorities allow certain requests (e.g. interactive) to be processed with privileges.
Publish-Subscribe System (PS)	Publishes the processed responses to channels named after the request ids. Everyone who is interested in the responses, especially the Ss, subscribe to the corresponding channels.
Workers (W)	Pop requests from the Qs and have them processed by their localhost APP. Update the C and publish the responses to the PS.
Applications (APP)	Get their requests from the localhost W and are developed using any existing WAF. Push the updated content to their W.
Load-Balancers (LB)	Receive the requests from the clients and distribute them to Ss.
Databases (DB)	Provide content for the access by the APPs.
Service Component	**Description**
Indexer	Initially fills C with all contents by emitting requests to the Qs.
Monitor	Watches Q-lengths and job-deadlines for resource provisioning.
Session	Registers, validates and destroys user sessions.
Access	Authorises resource access by user sessions.

III. Proposed Web Scaling Framework

The general concept of the proposed WSF named *Scales* is to cache all available content and distinguish each request to be either a read-request R_R or a request that needs processing R_P. R_Rs are directly served from cache and R_Ps are queued for processing. R_Rs never hit a web application which implicates that only requests that need processing are forwarded to a web application. Depending on the ratio between R_Ps and R_Rs this can drastically reduce the amount of requests that arrive at a web application, which improves the overall performance as the processing by the web application is slower than the read from a cache. Each of the components in Table I is a cluster by itself and fully horizontally scalable (Fig. 2).

A. Request Flow

Requests enter the system through an array of LBs. The LBs then forward each request to one of the system's Ss. The system has a read sub-system graph with the directed edges $S_R = \{(LB, S), (S, C), (C, S), (S, LB)\}$ and a processing sub-system graph $S_P = \{(LB, S), (S, Q), (Q, W), (W, A), (A, W), (W, PS), (PS, S), (S, LB)\}$. The S selects either S_R or S_P based on the HTTP method where $GET \to S_R$ and every other verb $\to S_P$. If S_R is selected, S gets the cached url path from a C, resolves possible fragments and responds

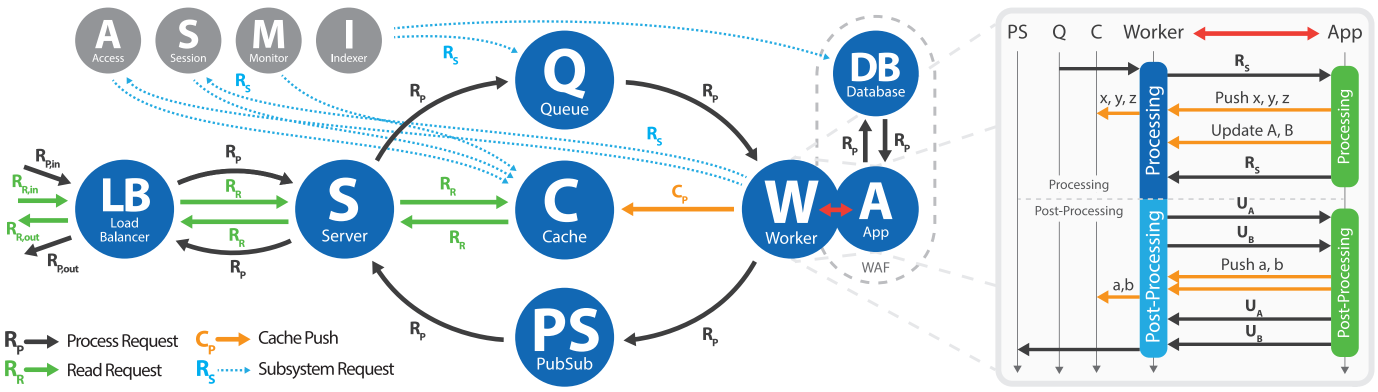

Fig. 2. **The flow of requests through the proposed prototype with a detail view of the processing and synchronous post-processing.**

with the result. If S_P is selected, S puts the request in a Q and waits for a response event from the PS. One of the Ws that has processing capabilities pops the request from Q and sends it to the APP on its localhost. APP processes the request, pushes the updated contents through W to C and declares its dependent url paths that need to be called by W before the request can return to PS. W collects the updates, optimises the dependencies and starts the post-processing by requesting all dependent update actions from the app. Once all updates are done and all new content is pushed to C, it emits the response to PS where S listens for the response-event and responds with the result.

B. Post Processing

The post-processing updates submitted by APP can be synchronous or asynchronous. *Synchronous* makes the worker emit the response after the update is processed (Fig. 2, R). *Asynchronous* emits the response regardless of the processing status of the update. This allows the application to guarantee updated resources at the time of the response where needed and speeds up response times for background updates.

IV. Modelling and Prototyping

The general concept of WSF is evaluated with a mathematical model. The model itself is evaluated with a prototype implementation of a WSF named scales.js and an abstract web application that allows testing different performance critical computations. The implementation is created using JavaScript with node.js and is able to work with WAFs of any language. The prototype (scales.js + abstract app) is then tested with different combinations of parameter sets. The results are compared to the calculated values from the model to derive the prediction errors. If the RMSE (Root-Mean-Square Error) is smaller than 5% of the maximum requests per second (RPS) the model is acceptable. Two versions of apps are considered: The normal app version V_N and the scaled app version V_S. In V_N requests are sent to a WAF directly, in V_S requests are sent to a WSF. The performance metrics are the average request duration D_N vs. D_S (lower is better) and the average requests per second RPS_N vs. RPS_S (higher is better). To be able to collect general data, the parameters that influence the performance of a web application are extracted. Each parameter tuple PT produces different performance metrics. The aim is to find the PTs where $RPS_S > RPS_N$ and $D_S < D_N$. The mathematical model describes the relationships between the PTs and the RPSs and Ds. An abstract app that behaves identical to a web application with the defined PTs is used to empirically validate the model.

A. Parameter Extraction

Table II lists the extracted parameters for a single-machine and a multi-machine cloud scope as the ability to scale horizontally is a key characteristic for cloud environments [13]. The performance is expected to be better for higher Cache/Processing Ratio (CPR) values because then, d_a decreasingly influences the response duration D. The tested CPR values are $CPR \in (1.0, 0.5, 0.0)$ where for V_S the tuples where $CPR = 1.0$ are expected to perform better and $CPR \in (0.5, 0.0)$ are expected to perform worse than V_N.

$$H_1 : RPS_S > RPS_N \; for \; \frac{1}{3} \; of \; all \; 81 \; PTs$$

With a correct setting of the system parameters PT_{system} the mathematical model is expected to predict with a root mean-squared error of less than 5%:

$$H_2 : RMSE < 5\% \; for \; all \; 81 \; PTs$$

B. Complex Network Delay

The previous extraction uses a constant value for the network delay. Whereas this simplification can be made for single-machine scenarios, in multi-machine scenarios a more complex model has to be used. An increasing concurrency also increases the network delay [17] until the network stack is completely loaded and not able to handle any more requests. For WSFs this means that the network delay for each component C_x is not a constant, but dependent on the concurrency c. The delay is specified as the minimum delay the network stack needs to compile, transmit and decompile the request $d_{n,min}$ plus the delay that is caused by increasing concurrency c:

$$d_{n,x} = d_{n,min} + \frac{c \cdot d_{n,gain}}{(c_{n,max} \cdot m_x) - c} \tag{1}$$

TABLE II
PARAMETERS OF THE PROPOSED WEB SCALING FRAMEWORK

Machine Parameter	**Description**
Cache/Processing Ratio: $CPR \in [1,0]$	The relation between GET and all other HTTP method type requests.
Response Size: $s \in [0,\infty]$ in kB	The size of the response body.
Concurrent Users: $c \in [1,\infty]$	The number of concurrent users or connections.
Post-Processing Updates: $u \in [0,\infty]$	The number of post-processing updates issued by a request.
Action Delay: $d_a \in [0,\infty]$ in s	The time it takes the WAF to process the request.
Network Delay: $d_n \in [0,\infty]$ in s	The minimum time it takes a request to travel the full network stack.
Size Delay: $d_s \in [0,\infty]$ in s	The time a single kilobyte adds to d_n.
Framework Delay: $d_f \in [0,\infty]$ in s	The time a WSF adds to the processing by traveling through S_P.
Cloud Parameter	**Description**
Component: $c_x \in \{C_N, C_S\}$	The components of $V_N : C_N = \{LB, APP\}$ and $V_S : C_S = \{LB, S, C, Q, W, APP, PS\}$ so C_{LB} denotes the load-balancer and C_S the server component.
Component Delay: $d_x \in [0,\infty]$ in s	The time a component needs to process a request. For the APP component this is the time the processing and post-processing takes.
Machine Size: $m_x \in [1,\infty], x \in \{C_N, C_S\}$	The number of machines used for a single component c_x. If a system uses four workers $m_W = 4$.
Machine Configuration: $mc_x, x \in \{V_N, V_S\}$ $mc_N = (m_{LB}, m_{APP})$, $mc_S = (m_{LB}, m_S, m_C, m_Q, m_W, m_{PS})$	A tuple representing the number of machines used per component. The tuple $mc_S = (1,2,1,1,4,1)$ uses two server-, four worker- and one of each of the other components.
Total Machine Count: $M \in [1,\infty]$	The total number of machines used for the system and thereby $M = \sum(mc_x), x \in \{V_N, V_S\}$.
Minimum Network Delay: $d_{n,min} \in [0,\infty]$ in s	The minimum network time it takes a request to flow through a component that is lost in the network stack.
Network Delay Gain: $d_{n,gain} > 0$	The distribution of how fast the delay is increased with growing concurrency. Low values add less delay for small c and rise extremely when $c \to c_{n,max}$. High values when $d_{n,gain} \to \infty$ increase the delay in a linear fashion.
Maximum Network Concurrency: $c_{n,max} > 1$	The maximum amount of requests where a component is not able to flow more requests through its network stack because it is fully loaded.

Each added machine m_x increases the possible maximum concurrency for component c_x as it brings in a new network stack.

C. Modelling for the Single-Machine Scope

The mathematical model predicts the average request duration D and the average requests per second RPS for V_N and V_S. Input to the model are the application specific parameter-tuples $PT_{app} = (CPR, d_a, s, u, c)$ and measured system parameters $PT_{system} = (d_n, d_s, d_f)$. Output is the predicted average value for the given PT that can be compared to the empirical data of the abstract app for evaluation purposes or current live system values.

1) Request Duration: Requests in V_N are always routed to the application. This means that every request is delayed by the action and the network:

$$D_N = d_a + d_n + (s \cdot d_s) \tag{2}$$

In V_S requests are routed to sub-system S_R or S_P in relation to the CPR. The delays for each sub-system are different: d_c for the time it takes to get a value from cache (2) and d_p for the time it takes the WSF to have the request processed by the app (3).

$$d_c = d_n + (s \cdot d_s) \tag{3}$$

$$d_p = d_n + (s \cdot d_s) + d_a + (d_a \cdot u) + d_f \tag{4}$$

Weighted by the CPR the duration is calculated with:

$$D_S = CPR \cdot d_c + (1 - CPR) \cdot d_p \tag{5}$$

2) Requests per Second: The RPS depend on the concurrency c of the incoming requests. The model makes no limitations on the value of c, however for real applications the value for c heavily depends on the app and can easily be determined empirically. For the prediction, the c value where the app was able to handle the most RPS should be chosen. If the app is able to handle c concurrent connections, the RPS for V_N is calculated with:

$$RPS_N = \frac{c}{D_N} \tag{6}$$

Analog to V_N this is also applicable for V_S:

$$RPS_S = \frac{c}{D_S} \tag{7}$$

D. Modelling for the Multi-Machine Cloud Scope

Instead of focussing on a request traveling the whole system, the request flow through the components is investigated. This allows to identify the slowest components as bottlenecks that prevent the other components from performing better. To consider all involved components, the measurement point for the maximum request flow per second $RFPS_{max}$ is located where the requests leave the system at R_{out} (Fig. 2). The value $RFPS_{max}$ for V_N and V_S depends on the used machines m_x for each component c_x. It specifies the theoretic maximum number of requests that leave the system per second. Compared to the previously calculated RPS_{max} where requests travel through a chain of component delays, the $RFPS_{max}$ only considers the faceless total number of requests that is given by the slowest component in the system.

1) Maximum Request Flow for Components: The request flow through a component c_x is influenced by the concurrency of requests c, the complex network delay $d_{n,x}$, the delay the component itself introduces for processing d_x and the size delay $s\dot{d}_s$. This allows a general calculation with the following equation:

$$RFPS_x = \frac{c}{d_{n,x} + d_x + (s \cdot d_s)} \tag{8}$$

To consider the post-processing of the W component, the W delay d_W is calculated like this:

$$d_W = d_a + (d_a \cdot u) \tag{9}$$

2) Total Maximum Request Flow: V_N uses the following components for all requests:

$$C_N = \{LB, A\} \tag{10}$$

The total maximum request flow is defined by the slower component of both:

$$RFPS_N = \min_{x \in C_N} \{RFPS_{N,x} \cdot m_x\} \tag{11}$$

For V_S both sub-systems S_R and S_P have to be considered with the following components:

$$C_{S,R} = \{LB, S, C\} \tag{12}$$
$$C_{S,P} = \{LB, S, Q, W, PS\} \tag{13}$$

For each sub-system the maximum request flow is defined like this:

$$RFPS_{S,R} = \min_{x \in C_{S,R}} \{RFPS_{S,x} \cdot m_x\} \tag{14}$$
$$RFPS_{S,P} = \min_{x \in C_{S,P}} \{RFPS_{S,x} \cdot m_x\} \tag{15}$$

This enables to calculate the total maximum request flow $RFPS_S$ considering the CPR:

$$RFPS_S = CPR \cdot RFPS_{S,R} + (1 - CPR) \cdot RFPS_{S,P} \tag{16}$$

3) Minimum Machines for Component: A key metric is the number of machines m_x that are needed for a component to satisfy a certain target $RFPS : T_x$. The calculation for that number of machines can be derived by equalising $RFPS_x = T_x$ and resolving it into m_x.

4) Total Machines: In V_N, the total number of machines needed to satisfy the target T_N is calculated by accumulating the machines needed for each component of C_N:

$$M_N = \sum_{x \in C_N} m_x(T_N) \tag{17}$$

For V_S, the target T_S is divided into the read sub-system $S_R : T_R$ and process sub-system $S_P : T_P$ by considering the CPR-weighted flow that is needed for both sub-systems:

$$T_{S,R} = CPR \cdot T_S \tag{18}$$
$$T_{S,P} = (1 - CPR) \cdot T_S \tag{19}$$

The total number of machines needed is then calculated by adding up the machines needed for both sub-systems:

$$M_S = \sum_{x \in C_{S,R}} m_x(T_{S,R}) + \sum_{x \in C_{S,P}} m_x(T_{S,P}) \tag{20}$$

V_S performs better than V_N if $M_S < M_N$ for a common target T.

5) Linear Total Machines Regression: The calculation of the total machine demand can be simplified with a linear regression approximation. The most interesting metric is the slope of the regression as it dictates the relation of the growth of the machine demand for an increasing $RFPS$. The slope for V_N can be calculated by determining the maximum $RFPS_{N,max}$

$$RFPS_{N,max} = \max_{x \in C_N} (RFPS_{N,x}) \tag{21}$$

and add up the number of machines needed for all components to reach it:

$$ms_N = \frac{\sum_{x \in C_N} \frac{RFPS_{N,max}}{RFPS_{N,x}}}{RFPS_{N,max}} \tag{22}$$

The full equation for the regression can then be compiled to:

$$M_{N,Reg} = \lceil ms_N + |C_N| \rceil \tag{23}$$

The regression for V_S needs to consider both sub-systems S_R and S_P. It needs to distinguish between shared $(C_{S,R,S}, C_{S,P,S})$ and not-shared $(C_{S,R,NS}, C_{S,P,NS})$ components of the two sub-systems:

$$C_{S,Shared} = C_{S,R} \cap C_{S,P} \tag{24}$$
$$C_{S,R,S} = C_{S,R} \cup C_{S,Shared} \tag{25}$$
$$C_{S,P,S} = C_{S,P} \cup C_{S,Shared} \tag{26}$$
$$C_{S,R,NS} = C_{S,R} \setminus C_{S,Shared} \tag{27}$$
$$C_{S,P,NS} = C_{S,P} \setminus C_{S,Shared} \tag{28}$$

In analogy to V_N, V_S the slope can be calculated with the maximum $RFPS_{S,max}$

$$RFPS_{S,max} = \max_{x \in C_S} (RFPS_{S,x}) \tag{29}$$

and the sum of the machines needed to reach it where the shared components are weighted by the CPR. To get the final regression slope both sub-system slopes (equations omitted due to length) are summed up weighted by the CPR:

$$ms_S = CPR \cdot ms_{S,R} + (1 - CPR) \cdot ms_{S,P} \tag{30}$$

The full equation for the regression of V_S is:

$$M_{S,Reg} = \lceil ms_S + |C_S| \rceil \tag{31}$$

6) Relative Average Machine Reduction: To compare the performance of V_N to V_S both slopes are set in contrast like this:

$$RAMR = (1 - \frac{ms_S}{ms_N}) \cdot 100 \tag{32}$$

The calculated value is the percentage of machine reduction V_S introduces compared to V_N. If $RAMR = 0$, both versions are likely to need the same amount of machines for the same $RFPS$. If $RAMR = 80$, V_S is expected to need 80% less machines than V_N. The bigger the $RAMR$, the greater the economic retrenchment of V_S.

TABLE III
ACTIONS OF THE PROTOTYPE

Method Path	Description
GET /items/:s/:d_a	A response with skBs is created and returned non blocking after d_a seconds. For V_S this action is fully served from cache.
POST /items/:s/:d_a/:u	Also returns a response with skBs, but the total delay is calculated $d_a + d_a * u$ because in V_S the request is delayed until all updates are post-processed.

E. Empirical Prototyping

To validate the mathematical model for the machine scope empirically, an implementation of a web application is needed. The web application needs to consider the different parameter-tuples it is configured with - e.g. it needs to delay the response by 0.5s if $d_a = 0.5$ and return a response of 100kB if $s = 100$.

V_N evaluates with parameter-tuples in the form $PT_N = (CPR, d_a, s)$ and V_S with $PT_S = (CPR, d_a, s, u)$ because u has no influence on V_N. Data is collected with all combinations from $CPR \in (1.0, 0.5, 0.0)$, $d_a \in (0.0, 0.5, 1.0)$, $s \in (25, 50, 100)$ and $u \in (0, 5, 10)$ which are typical parameters for popular web services [15]. V_N has 3 3-tuples which makes $3^3 = 27$ tuples and V_S has 4 3-tuples resulting in $3^4 = 81$ parameter-tuples. All other parameters need to be configured by either measuring the delays on the real system or monitoring values on the live system - e.g. the CPR. Table III shows the actions of the prototype that is implemented using the WAF node.js.

The data is collected for each of the parameter-tuples in V_N and V_S. A single test-run for a PT_x requests for a 10 minute period with an increasing concurrency from 0...100 over the full period. The maximum RPS_{max} and D_{max} is the result-pair. The application always runs on a freshly booted virtual machine (*2,6 GHz i7, 1,7 GB RAM, Ubuntu Server 13.04*). The results can then be compared to the predicted result-pairs calculated in the previous section.

V. RESULTS

In the first run, the parameter-tuples for V_N are tested. The maximum $RPS_{max,PT,x}$ and $D_{max,PT,x}$ over the 10 minute period is collected and set to be the performance baseline with a performance of 100%. Then the same tests are executed for V_S. The results are compared to the result of V_N which gives the final performance metric for each parameter-tuple.

A. Empirical Data

For 30 of 81 PTs (37%) the performance of V_S is better than the performance of V_N which allows to accept H_1. The interesting metric is $\Delta RPS = RPS_S - RPS_N$ which is positive if V_S is better and negative if V_N is better. The mean of all $\overline{\Delta RPS} = \mu = 54$ which means that over all PTs V_S was able to generate $54RPS$ more than V_N. The mean of the 30 PTs that performed better is $\mu_{better} = 3952$, the mean of the worse PTs is $\mu_{worse} = -2239$.

B. Model Evaluation

The RMSE is calculated based on the predicted versus the empirical RPS. With a $d_n = 0.01$ the absolute maximum $RPS_{max} = 10000$. This allows to accept H_2 if $RMSE \leq RPS_{max} \cdot \alpha \leq 10000 \cdot 0.05 \leq 500$. With the system parameters $PT_{system} = (d_n, d_s, d_f) = (0.01, 0.0001, 0.065)$ the $RMSE = 232 < 500$ which allows to accept H_2 and therewith the mathematical model as an approximative performance prediction on a single machine. The empirical evaluation of the model for the cloud scope using many machines is the next planned step in the future work.

C. Analytical Effect of Adding Machines

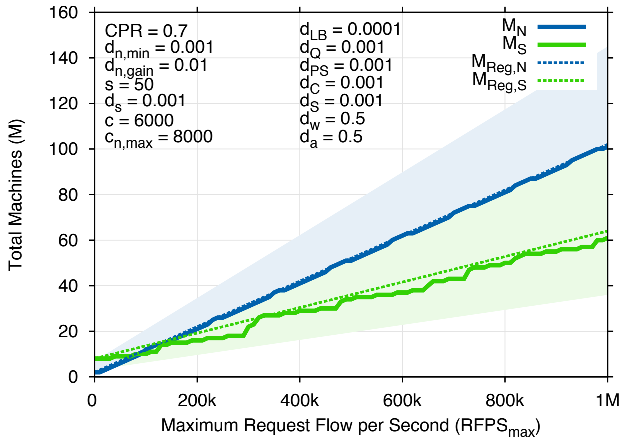

Fig. 3. **Total Machine comparison (prediction and linear regression).** V_S **constantly needs** 44.44% **fewer machines than** V_N (lower is better)

Fig. 3 shows a comparison of the machines needed for V_S and V_N with the growing target $RFPS : T$. V_S performs better than V_N because fewer machines are needed for the same target. The initial better performance of V_N is derived from the minimum number of machines for each system as - by definition - each component needs at least one dedicated machine. The minimum number of machines for $V_N : |C_N| = 2$ and for $V_S : |C_S| = 6$. These limits have a mathematical reason, in application however multiple components can be run on a single machine for small systems.

For the parameters of Fig. 3 the linear regression approximates slopes of $ms_N = 10 \cdot 10^{-5}$ and $ms_S = 5.6 \cdot 10^{-5}$. The $RAMR = 44.44$ which means that, as the RFPS increases, V_S constantly needs 44.44% fewer machines than V_N. The solid area indicates the scope of the regression with respect to possible CPR values. The lower border is the total number of machines if full caching is used ($CPR = 1, ms = 2.7 \cdot 10^{-5}$), and the upper border if every request needs to be processed ($CPR = 0, ms = 12.3 \cdot 10^{-5}$). The green area indicates the CPR values where V_S needs fewer machines than V_N. In summary this means that WSFs are economically efficient because they provide a fine grained scaling control that only scales the necessary bottleneck components. WAFs in contrary only allow to scale the whole system - independent of the bottleneck. This also scales unneeded resources that are dispensable but costly.

VI. Application

How can the derived model be used to predict performance of real systems? Consider three sample web applications (S_1, S_2, S_3) created with different WAFs (JEE, PHP, Ruby on Rails). All of them are already deployed in production and as a scaling solution a WSF is considered. The first step to calculate the predicted performance is to gather the app- $PT_{app} = (CPR, d_a, s, u, c)$ and system-parameters $PT_{system} = (d_n, d_s, d_f)$ by measuring the production system.

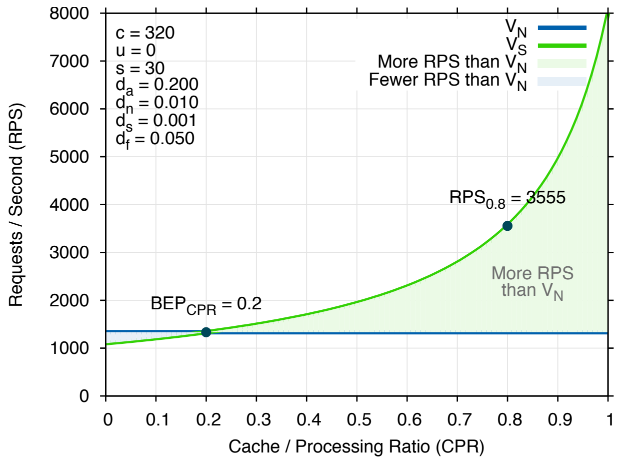

Fig. 4. CPR **dependent** RPS **development for** S_2**. The** CPR **where** V_S **starts to perform better than** V_N **is given by break-even** $BEP_{CPR} = 0.2$**. At** $CPR = 0.8$**,** V_S **generates** 166.6% **more** RPS **than** V_N**.**

A. Calculations for the Single-Machine Scope

Assume the same $PT_{system} = (0.01, 0.001, 0.05)$ for all apps and the PT_{app} : S_1 : (0.7, 0.3, 65, 3, 60), S_2 : (0.8, 0.2, 30, 0, 320), S_3 : (0.9, 0.1, 83, 2, 20). The RPS calculation then predicts the following performance for (S_1, S_2, S_3): RPS_S : (181, 3555, 162) and RPS_N : (160, 1333, 104). The comparison of $(RPS_S/RPS_N) - 1$ then predicts the performance to increase by $(13.1\%, 166.6\%, 55.7\%)$ using a WSF on a single machine. Fig. 4 illustrates the highest performance increase of S_2.

B. Calculations for the Multi-Machine Cloud Scope

Assume the same target $RFPS_T = 100k$ for all apps. The M_{Reg} calculation predicts $M_{Reg,N}$ = (723, 84, 1343) and $M_{Reg,S}$ = (463, 60, 1260) machines to satisfy the target. The $RAMR$ predicts a reduction by $(36.5\%, 35.1\%, 6.5\%)$ machines using a WSF in a cloud environment.

VII. Conclusion and Future Work

We have proposed WSF, a novel automated scaling layer, for highly scalable web services. Both WAFs and developers could then focus on creating the business logic layer of web services. The proposed mathematical model allows a universally applicable prediction of performance in the single-machine- and multi-machine scope. The implemented prototype serves as a general composition architecture and demonstrates both the feasibility and possible performance increase of such a framework.

The results showed that the application of a WSF can triple the performance of a single machine and additionally reduce the number of total machines by 44%. In our future work we will create and deploy a WSF to the mobile cloud [1]–[3]. The research needed for the shift to the mobile cloud involves the inspection of distributed runtime environments, offloading policies, distributed routing technologies and many more. We hope to benefit from the ongoing research in those fields and incorporate the results in our future work.

References

[1] G. Huerta-Canepa and D. Lee, "A virtual cloud computing provider for mobile devices", p. 6, 2010.
[2] M. V. Pedersen and F. H. P. Fitzek, "Mobile Clouds: The New Content Distribution Platform", Proc. IEEE, vol. 100, no. Special Centennial Issue, pp. 1400-1403, 2012.
[3] N. Fernando *et al.*, "Mobile cloud computing: A survey", Future Generation Computer Systems, vol. 29, no. 1, pp. 84-106, Jan. 2013.
[4] H. Han *et al.*, "Cashing in on the Cache in the Cloud", Parallel and Distributed Systems, IEEE Transactions on, vol. 23, no. 8, pp. 1387-1399, 2012.
[5] D. Chiu and G. Agrawal, "Evaluating caching and storage options on the amazon web services cloud", pp. 17-24, 2010.
[6] Y. Lu et al., "Join-Idle-Queue: A novel load balancing algorithm for dynamically scalable web services", Performance Evaluation, vol. 68, no. 11, pp. 1056-1071, Nov. 2011.
[7] N.-L. Tran *et al.*, "EQS: An Elastic and Scalable Message Queue for the Cloud", presented at the Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on, 2011, pp. 391-398.
[8] M. Mao and M. Humphrey, "Auto-scaling to minimize cost and meet application deadlines in cloud workflows", pp. 1-12, 2011.
[9] A. Wolke and G. Meixner, "Twospot: A cloud platform for scaling out web applications dynamically", pp. 13-24, 2010.
[10] W. Lu *et al.*, "AzureBlast: a case study of developing science applications on the cloud", pp. 413-420, 2010.
[11] P. Marshall *et al.*, "Elastic Site: Using Clouds to Elastically Extend Site Resources", presented at the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, pp. 43-52.
[12] T. Tung *et al.*, "Highly Resilient Systems for Cloud", presented at the 2012 IEEE 19th International Conference on Web Services (ICWS), pp. 678-680.
[13] M. Armbrust *et al.*, "A view of cloud computing", Commun. ACM, vol. 53, no. 4, pp. 5058, Apr. 2010.
[14] L. M. Vaquero *et al.*, "Dynamically scaling applications in the cloud", ACM SIGCOMM Computer Communication Review, vol. 41, no. 1, pp. 45-52, 2011.
[15] B. Kahle, "HTTP Archive", [Online]. Available: http://httparchive.org, Jan. 2014.
[16] J. Idziorek, "Discrete event simulation model for analysis of horizontal scaling in the cloud computing model", pp. 3004-3014, 2010.
[17] L. Kleinrock, "The latency/bandwidth tradeoff in gigabit networks", IEEE Communications Magazine, vol. 30, no. 4, Apr. 1992.