2018 IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems

TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research

Jóakim v. Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, Samuel Kounev
University of Würzburg
{joakim.kistowski, simon.eismann, norbert.schmitt, andre.bauer, johannes.grohmann, samuel.kounev}@uni-wuerzburg.de

***Abstract*—Modern distributed applications offer complex performance behavior and many degrees of freedom regarding deployment and configuration. Researchers employ various methods of analysis, modeling, and management that leverage these degrees of freedom to predict or improve non-functional properties of the software under consideration. In order to demonstrate and evaluate their applicability in the real world, methods resulting from such research areas require test and reference applications that offer a range of different behaviors, as well as the necessary degrees of freedom.**

Existing production software is often inaccessible for researchers or closed off to instrumentation. Existing testing and benchmarking frameworks, on the other hand, are either designed for specific testing scenarios, or they do not offer the necessary degrees of freedom. Further, most test applications are difficult to deploy and run, or are outdated.

In this paper, we introduce the TeaStore, a state-of-the-art micro-service-based test and reference application. TeaStore offers services with different performance characteristics and many degrees of freedom regarding deployment and configuration to be used as a benchmarking framework for researchers. The TeaStore allows evaluating performance modeling and resource management techniques; it also offers instrumented variants to enable extensive run-time analysis. We demonstrate TeaStore's use in three contexts: performance modeling, cloud resource management, and energy efficiency analysis. Our experiments show that TeaStore can be used for evaluating novel approaches in these contexts and also motivates further research in the areas of performance modeling and resource management.

I. Introduction

Modern distributed component and/or service-based applications have complex performance characteristics, as the constituent services feature different bottlenecks that may even change over time, depending on the usage profile. However, these applications also offer many degrees of freedom, which are intended to help deal with these challenges. They can be deployed in various ways and configured using different settings and software stacks. These degrees of freedom can be used at design-time, deployment-time, and at run-time for continuous system optimization. Current research employs many methods of analysis, modeling, and optimization that utilize these degrees of freedom at different points of the software life-cycle to tackle the challenging performance behavior [1], [2]. More general, the goal of such research is the improvement of a running system's non-functional properties and may include dependability [3], [4], or energy efficiency [5], [6], [7].

Verifying, comparing, and evaluating the results of such research is difficult. To enable practical evaluation, researchers need a software application (1) that they can deploy as reference and (2) that offers realistic degrees of freedom. The reference application must also feature sufficient complexity regarding performance behavior to warrant optimizing it in the first place. Finding such an application and performing the necessary experiments is often difficult. The software in question should be open source, available for instrumentation, and should produce results that enable analysis and comparison of research findings, all while being indicative of how the evaluated research would affect applications in production use.

Real world distributed software is usually proprietary and cannot be used for experimentation. It is often inaccessible and lacks the potential for instrumentation. In addition, evaluations conducted using such software are difficult to reproduce and compare, as the software used remains inaccessible for other researchers. Existing test and reference software, on the other hand, is usually created for specific testing scenarios [9]. It is often designed specifically for evaluating a single contribution, which makes comparisons difficult. Other existing and broadly used test software does not offer the necessary degrees of freedom and is often manually adapted [10]. Some of the most widely used test and reference applications, such as RUBiS [11] or Dell DVD Store [12], are outdated and therefore not representative of modern real world applications. Newer distributed reference applications, such as Sock Shop [13], are built for maximum scalability and consistent performance and do not pose the performance challenges that current research aims at.

In this paper, we introduce TeaStore[1][2], a micro-services-based test and reference application that can be used as a benchmarking framework by researchers. It is designed to provide multiple degrees of freedom that researchers can vary when evaluating their work. TeaStore consists of five different services, each featuring unique performance characteristics and bottlenecks. Due to these varying performance characteristics and its distributed nature, TeaStore may also be used as a software for testing and evaluation of software

[1]TeaStore on GitHub: https://github.com/DescartesResearch/TeaStore/
[2]TeaStore on DockerHub: https://hub.docker.com/u/descartesresearch/

2375-0227/18/$31.00 ©2018 IEEE
DOI 10.1109/MASCOTS.2018.00030

TABLE I: Micro-service Benchmark [8] and our Research Benchmark Requirements for the TeaStore in comparison to ACME Air, Spring Cloud Demo Apps, Shocks Shop and MusicStore.

Micro-service Benchmark Requirement		**TeaStore**	**ACME Air**	**Spring Cloud Demo**	**Sock Shop**	**MusicStore**
R1	Explicit Topological View	✓		✓	✓	
R2	Pattern-based Architecture	✓	✓	✓	✓	✓
R3	Easy Access from a Version Control Repository	✓	✓	✓	✓	✓
R4	Support Continuous Integration	✓		✓	✓	
R5	Support for Automated Testing	✓		✓	✓	
R6	Support for Dependency Management	✓	✓	✓	✓	✓
R7	Support for Reusable Container Images	✓	✓	✓	✓	
R8	Support for Automated Deployment	✓			✓	
R9	Support for Container Orchestration	✓	✓		✓	
R10	Independence of Automation Technology	✓			✓	
R11	Alternate Versions		✓			✓
R12	Community Usage & Interest		✓			
Research Benchmark Requirement						
B1	Service must Stress System Under Test	✓	✓	(unknown)		✓
B2	Support for Different Load Behavior in Services	✓	✓	✓		
B3	Support for Different Load Generators	✓	✓	✓	✓	✓
B4	Load Profiles Publicly Available	✓				

performance models and model extraction techniques. It is designed to be scalable and to support both distributed and local deployments. In addition, its architecture supports run-time scalability as services and service instances can be added, removed, and replicated at run-time. The services' different resource usage profiles enable performance and efficiency optimization with non-trivial service placement and resource provisioning decisions.

To summarize, we envision the use of TeaStore in the following research areas, among others:

1) Evaluation of software performance modeling approaches, model extractors, and model learners.
2) Evaluation of run-time software performance management techniques such as auto-scaling and service placement algorithms.
3) Evaluation of software energy efficiency, power models, and optimization techniques.

We demonstrate the applicability of TeaStore as a test application and benchmarking framework by using it as a reference software in experiments that show its applicability in each of the three motivating research areas. We show that TeaStore can be used as a reference scenario for performance modeling by creating a simple performance model to predict the application performance for different deployment options. This example model also illustrates the limitations of simplified software performance models for predicting the performance of complex distributed applications, highlighting open research challenges. In addition, we show TeaStore's elastic run-time scalability by running it using a state-of-the-art baseline auto-scaler. We show that the baseline auto-scaler can scale TeaStore elastically at run-time, while also demonstrating the limitations of conventional auto-scalers for complex applications. Finally, we examine the energy efficiency and power consumption when scaling TeaStore over multiple physical hosts. We show that distribution and placement decisions lead to different power and energy efficiency behavior, which can be used to evaluate energy optimization methods.

II. Related Work

With the emergence of modern trends like DevOps [14], the focus of research benchmarks has moved from fixed multi-tier application benchmarks, like SPECjEnterprise [15], towards more scalable micro-service applications. With this shift in how applications are developed, deployed and maintained, requirements for research benchmarking reference applications have changed.

Micro-services offer many degrees of freedom, such as placing your services in a public or private cloud [16], predictive elastic resource scaling [17], and auto-scaling [18]. Additional requirements become necessary for a reference application to be used in research. Aderaldo et al. [8] identify 15 requirements for micro-service research benchmarks. Table I lists these requirements and checks common benchmark applications for compliance. In terms of compliance to these criteria, the TeaStore satisfies all criteria except *Alternate Versions* (R11) and *Community Usage & Interest* (R12) and is, in terms of requirements, identical to the Sock Shop [13]. R12, in particular, cannot be satisfied by a newly proposed reference application. ACME Air, Spring Cloud Demo and MusicStore [19], [20], [21], also compared in Table I, satisfy fewer requirements. These criteria suit micro-service benchmarks, yet they do not cover the ability of an application to be used as a reference research benchmark application in research domains, such as resource management and software analysis and modeling. We therefore extended the requirements by four research benchmark requirements B1 - B4 in Table I. To research current performance challenges, an application must put actual load on the System Under Test (B1), without focusing on a single server component like memory or CPU load (B2). The application should also be able to be put under load by different load generators to fit a wide variety of benchmarking environments (B3) and the used load profiles should be public for repeatability (B4). The TeaStore satisfies all these criteria while the next best application, ACME Air, only misses publicly available load profiles.

With the changing requirements and the fast living development of modern web services, the available reference applications cannot keep up and thus cannot cover all requirements for micro-service research benchmarking. The Rice University Bidding System (RUBiS) was first released in 2002. It is an eBay-like bidding platform [11] that has been implemented using various technologies including PHP, EJB and Java HTTP servlets. The different available implementations allow for benchmarking of the underlying technologies [22], the specific implementations themselves [23], the impact of applying common design patterns [24] and the already mentioned predictive elastic resource scaling methods [17]. RUBiS only has a single application service and an Apache HTTP load balancer [11]. While RUBiS supports remote procedure calls across multiple hosts, it can only be scaled as a single service. In contrast, each service of the TeaStore can be scaled individually. This allows for creating different resource usage characteristics through the deployment of the TeaStore's services, consistent with the micro-service paradigm. Similar to RUBiS, the Dell DVD Store [12], released in 2001, also features multiple implementations. It is a single-service application that implements a simple web store for DVDs. The CloudStore application on the other hand, has a different focus. It is a book store built as a reference application for comparing cloud providers, cloud service architectures, and cloud deployment options. It implements the Transaction Performance Council's TPC-W specification. However, TPC-W is obsolete since 2005. Like RUBiS, CloudStore suffers from inherent scalability bottlenecks. It has been used for elasticity benchmarking, evaluating scalability metrics [25], [26], [27] in order to test an infrastructure's ability to mitigate these bottlenecks. Just like RUBiS and the DVD Store, the CloudStore application lacks many degrees of freedom regarding deployment and configuration due to their single-service implementations.

A more modern and established benchmark both in research and commercial domains is the SPECjEnterprise2010 [15] from the Standard Performance and Evaluation Corporation (SPEC). SPECjEnterprise2010 implements a three tier web store with separate UI, business logic, and persistence components. It can be used to evaluate resource management techniques in a distributed setting [10]. Unfortunately, it lacks support for modern micro-service architectures due to its classic three tier architecture. It therefore can also not fulfill the micro-service requirements discussed earlier.

The PCM Media Store is designed as a component-based application with components that can be deployed on different systems and therefore closer to a micro-service application than RUBiS, Dell DVD Store, CloudStore and SPECjEnterprise2010. Each component can be replaced to support different architectures. The Media Store is specifically developed for evaluating design-time performance modeling techniques [9]. Also developed with its focus on design-time performance modeling is the Common Component Modeling Example (CoCoME). It was used in a Dagstuhl Research Seminar to compare different modeling approaches [28]. TeaStore, on the other hand, not only supports design-time modeling, but is also available with built-in instrumentation required for benchmarking, run-time model extraction, as well as elasticity and energy-efficiency measurements.

Many other benchmark and test applications, like JPetStore [29], PetClinic [30], ACME Air [19], Spring Cloud Microservice Example [20], Sock Shop [13] and MusicStore [21] are available. Yet, PetClinic, SCME, Sock Shop and MusicStore are primarily designed as demonstrators for specific technologies and not as a research benchmark reference application. Additionally, modern services such as Sock Shop are built with consistent performance in mind and do not pose the performance challenges that current research aims at.

The TeaStore offers a modern micro-service architecture for research benchmarks compared to other reference and benchmarking applications. It can also fullfill most micro-service benchmarking requirements shown in Table I without beeing a technology demonstrator like Sock Shop. The TeaStore also offers different performance characteristics for each service and is not limited to a single use case like the PCM Media Store and CoCoME.

III. The TeaStore

The TeaStore is an online store for tea and tea related utilities. Its products are sorted into categories. For online shopping, the store supports an overview of products including preview images for each category and featuring a configurable number of products per page. All pages of the TeaStore show an overview header bar and include the category menu and page footer. As main content, it shows the products for the selected category, including shortened product information and the preview image. Depending on the number of products shown per page, the user has the option to cycle through multiple pages of the category view.

Each product can be viewed on a separate product page containing detailed information, a large image, and advertisements for other store items. Besides the regular header, footer, and category list, this page includes a detailed image of the product (provided by the Image Provider Service), a description, and price. The page also contains an advertisement panel suggesting three products that the user might be interested in. The advertised products are provided by the Recommender Service and are selected depending on the viewed product.

All products can be placed in a shopping cart and users can proceed to order the current shopping cart. The user can choose to modify the shopping cart at any time. The shopping cart page lists all products currently included in the cart together with some product information and the quantity. The shopping cart view also displays product advertisements, which are, again, provided by the separate Recommender service and selected depending on the shopping cart's contents.

To order, the user must supply personal information about the billing address and payment details. After confirmation by the user, the current shopping cart is stored in the order history database through the Persistence service. The store also supports user authentication and login. Registered users can view their order history after login.

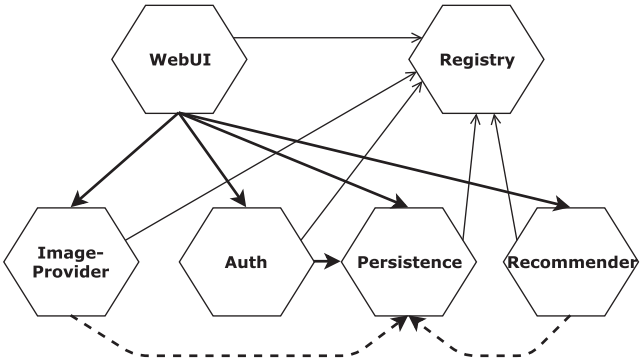


Fig. 1: TeaStore Architecture.

In addition to regular operations, the TeaStore's user interface provides an overview of all running service instances and an option to regenerate the database. In case a specific database setup or size is necessary, it can be regenerated with user defined parameters. These include the number of categories, number of products per category, number of users, and maximum orders per user history. The service overview and database regeneration are not intended to be run during an experiment run, but separately on experiment setup.

All functionality is contained within the five primary micro-services and the Registry service.

A. Architecture

The TeaStore consists of five distinct services and a Registry service as shown in Figure 1. All services communicate with the Registry. Additionally, the WebUI service issues calls to the Image-Provider, Authentication, Persistence and Recommender services.

The Image provider and Recommender both connect to a provided interface at the Persistence service. However, this is only necessary on startup (dashed lines). The Image provider must generate an image for each product, whereas the Recommender needs the current order history as training data. Once running, only the Authentication and the WebUI access, modify, and create data using the Persistence.

All services communicate via representational state transfer (REST) calls, as REST has established itself as the de-facto industry standard in the micro-service domain. The services are deployed as web-services on Apache Tomcat. Yet, the services can be deployed on any Java application server able to run web-services packaged as `war` files. As an alternative to deploying the `war` files, we provide convenient Docker images, containing the entire Tomcat stack. Each service is packaged in its own `war` file or Docker image.

The TeaStore uses the client-side load balancer Ribbon[3], to allow replication of instances of one service type. Ribbon distributes REST calls among running instances of a service. Instead of using Netflix Eureka[4], the TeaStore uses its own registry that supplies service instances with target instances of a specified target specific service type. To enable this, all running instances register and unregister at the registry, which can be queried for all running instances of a service. This allows for dynamic addition and removal of service instances during run-time. Each service also sends heartbeats to the registry. In case a service is overloaded or crashed and therefore fails to send heartbeat messages, it is removed from the list of available instances. Subsequently, it will not receive further requests from other services. This mechanism ensures good error recovery and minimizes the amount of requests sent to unavailable service instances that would otherwise generate request timeouts.

As the TeaStore is primarily a benchmarking and testing application, it is open source and available to instrumentation using available monitoring solutions. Pre-instrumented Docker images for each service that include the Kieker[5] monitoring application [31], [32] as well as a central trace repository service, are already available. We choose Kieker, as it requires no source code instrumentation and the instrumentation can be adapated at runtime. However, as the TeaStore is open source, other monitoring solutions, such as Prometheus [6] or Logstash [7] can also be utilized.

Generally, all requests to the WebUI by a user or load generator are handled in a similar fashion. The WebUI always retrieves information from the Persistence service. If all information is available, images for presentation are fetched from the Image provider and embedded into the page. Finally a Java Server Page (JSP) is compiled and returned. This behavior ensures that even non-graphical browsers and simple load generators that otherwise would not fetch images from a regular site cause image I/O in the TeaStore, ensuring comparability regardless of the load generation method.

Figure 2 shows the service calls for a user request for a product information page. After receiving the HTTP request, the WebUI checks the user's login status by calling the Auth service. Next, it queries the Persistence for the corresponding product information, based on a unique identifier. Afterwards, the WebUI requests advertisement options for the current product from the Recommender, which generates a recommendation based on the learned historical order data. The call to the Recommender takes the current login status into account. Specifically, a logged in user receives personalized recommendations, whereas an anonymous user is served recommendations based on general item popularity. Having received all product information, the WebUI queries the image provider to supply a full size image of the product shown in detail and preview images for the recommendations. The image data is embedded in the HTML response as base-64 encoded strings.

[3]Netflix Ribbon: https://github.com/Netflix/ribbon
[4]Netflix Eureka: https://github.com/Netflix/eureka

[5]Kieker APM: http://kieker-monitoring.net/
[6]Prometheus: https://prometheus.io/
[7]Logstash: https://www.elastic.co/products/logstash

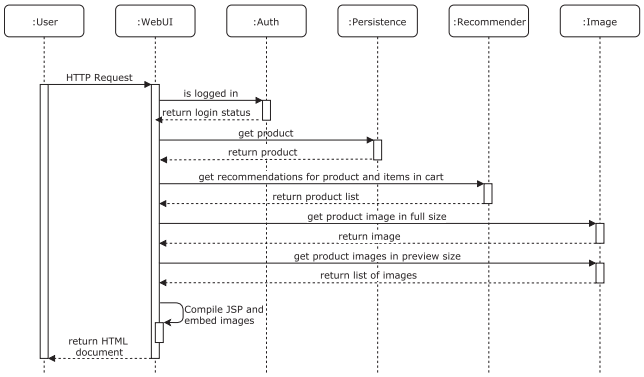


Fig. 2: Service calls when requesting product page.

B. Services

The TeaStore consists of five services, in addition to a registry necessary for service discovery and load balancing. In case monitoring is enabled, a trace repository service can be used to collect the monitoring traces centrally.

1) WebUI: This service provides the user interface, compiling and serving Java Server Pages (JSPs). All data, available categories, their products, product recommendations and images, are retrieved from the Image provider and Persistence service instances. The WebUI service performs preliminary validity checking on user inputs before passing the inputs to the Persistence service. The WebUI focuses purely on presentation and web front-end operations. However, the performance of the WebUI depends on the page that has to be rendered as each page contains at least one picture in different formats.

2) Image Provider: The Image provider serves images of different image sizes to the WebUI when being queried. It optimizes image sizes depending on the target size in the presentation view. The Image provider uses an internal cache and returns the image with the target size from the cache if available. If the image is not available for this size, the image provider uses the largest available image for the category or product, scales it to the target size, and enters it into the cache. It uses a least frequently used cache, reducing resource demand on frequently accessed data. Through the caching, the response time for an image depends on whether this image is in the cache or not. This service queries the Persistence service once on start-up to generate all product images with a fixed random seed.

3) Authentication: This service is responsible for the verification of both the login and the session data of a user. The session data is validated using SHA-512 hashes. For login verification, the BCrypt algorithm is used. The session data includes information about the current shopping cart content, the user's login status and old orders. Thus, the performance of the hashing for the session data depends on number of articles in the cart and number of old orders. Furthermore, as all session data is passed to the client, the Authentication itself manages to remain stateless and does not need additional information on startup.

4) Recommender: The Recommender service uses a rating algorithm to recommend products for the user to purchase. The recommendations are based on items other customers bought, on the products in a user's current shopping cart, and on the product the user is viewing at the time. The initial Recommender instance usually uses the automatically generated data-set, as provided by the persistence service at initial startup, for training. Any additional Recommender instance queries existing Recommender service instances for their training data-set and uses only those purchases for training. This way, all Recommenders stay coherent, recommending identical products for the same input. In addition, using identical training input also ensures that different instances of the Recommender service exhibit the same performance characteristics, which is important for many benchmarking and modeling contexts. The Recommender service queries the Persistence service only once on startup.

For recommending, different algorithm implementations exhibiting different performance behaviors are available. Next to a fallback algorithm based on overall item-popularity, two variants of Slope One [33] and one order-based nearest-neighbor approach are currently implemented. One variant of Slope One calculates the predicted rating matrix beforehand and keeps it in the memory (memory-intensive), wheras the other one calculates every row if needed, but discards all results after each recommendation step (CPU-intensive).

5) Persistence: The Persistence service provides access and caching for the store's relational database. Products, their categories, purchases, and registered store users are stored in a relational SQL database. The Persistence service uses caching to decrease response times and to reduce the load on the database itself for improved scalability. The cache is kept coherent across multiple Persistence service instances. We use the EclipseLink JPA implementation as a black-box cache. All data inside the database itself is generated at the first start of the initial persistence instance. By using a persistence service in separation from the actual database, we improve scalability by providing a replicable caching service. However, the performance of the database accesses depends on the content in the database that is changed or can be repopulated during the operation of the store.

6) Registry: The registry is not part of the TeaStore application under test but is a necessary support service. It keeps track of all running service instances, their IP addresses or host names and port numbers under which the services are available. All service instances send keep-alive messages to the registry after registration. If a service unregisters or no keep-alive message is received within a fixed time frame, the service is removed from the list of available service instances. All services can query the list of service instances for a specified service type in order to distribute their outgoing requests between running target instances.

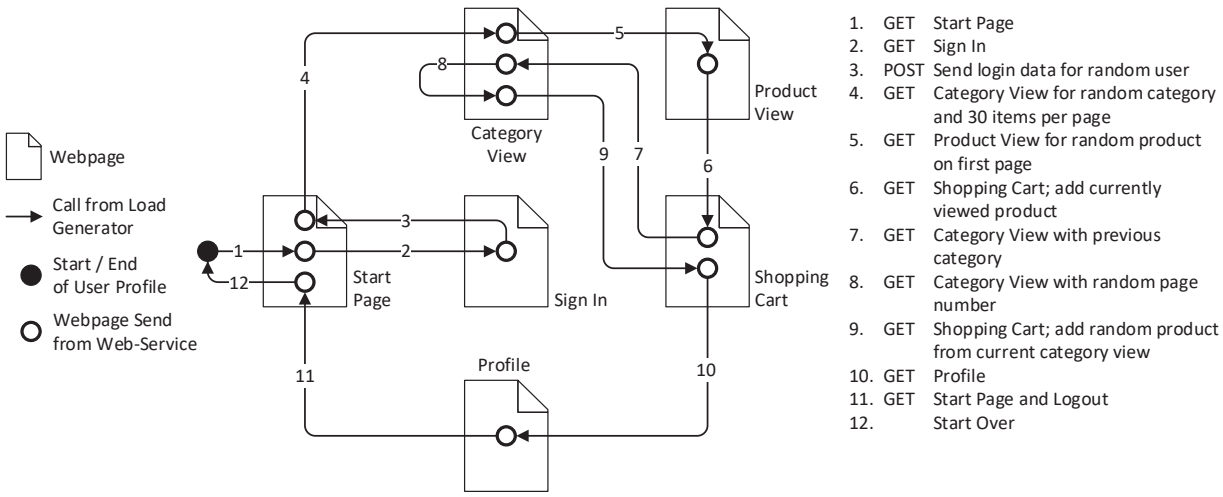


Fig. 3: "Browse" user profile configured in HTTP load generator for our use-cases, including web pages delivered and HTTP request type. Unused web pages are omitted for clarity.

7) TraceRepository: The services are configured with optional Kieker monitoring[8] [31], [32]. With monitoring enabled, each service instance collects information about utilization, response times and call paths. Collecting these monitoring traces manually is only feasible for small deployments. Therefore, we offer a central trace repository, which consists of an AMQP server coupled with a graphical web interface. All service instances send their logs to the AMQP server. The web interface collects them and makes them available for download. The trace repository does not only reduce the effort required to acquire the monitoring traces, but also enables online analysis such as online resource demand estimation [34]. Kieker traces are also available for use with tools other than Kieker's own tooling, as they can be automatically transformed to Open Execution Trace Exchange (OPEN.xtrace) traces, an open source trace format enabling interoperability between software performance engineering approaches [35].

IV. Use-Cases

We show the TeaStore's use in three use-cases. All use-cases are designed to show that the TeaStore can be used as a software that exhibits the characteristics that are addressed or needed for the respective field of research. Specifically, we show the TeaStore's use in three areas of research. We show that it can be used as a distributed application for evaluating and extracting software performance models, for testing single and multi-tier auto-scalers, and for software energy-efficiency analysis and management. The use-cases are constructed as examples of the state-of-the art that current research has to compare itself against. E.g., the auto-scaling example uses the standard state-of-the-art reactive auto-scaler, against which more advanced research scalers have to be compared.

[8]Kieker setup: https://github.com/DescartesResearch/TeaStore/wiki/Testing-and-Benchmarking

For all of the experiments, we utilize much of the standard testing tools and profiles found in the TeaStore's documentation. We generate load using an HTTP load generator, which sends requests based on an open workload model, and was first introduced in [36]. Requests are defined using a load profile, in which the request rate may vary over time. For our measurements, the load profile may be constant, linearly increasing (stress test profile), or based on a real-world load trace. We extract the real world trace using the LIMBO load intensity model extraction mechanism [37] and modify it to describe the load intensity in a range that can be handled by our system under test (SUT), varying between almost no load and the maximum throughput capacity. On the time scale, the real-world traces are modified to execute the load variations of the original multiple-day-long trace within one hour.

The content of the requests (meaning, the user actions) are defined using a stateful user profile. Each time a request is sent, an idle user from the pool of users is selected to execute a single action on the store. The user then performs that action and returns to the pool. This means that the user state and actions are chosen as they would be in a closed workload model [38], whereas the arrival times of the single requests are chosen according to an open workload model. We use a cyclical user profile, in which users browse the store. Figure 3 shows this profile. Users log in, browse the store for products, add these products to the shopping cart and then log out. The number of users is chosen depending on the maximum load.

We place the TeaStore's primary services on several HPE ProLiant DL160 servers, each equipped with a Xeon E5-2640 v3 processor with 16 logical cores at 2.6 GHz and 32 GB RAM. The servers run Debian 9 and Docker 17.12.1-ce. The service registry is executed on an additional physical host. We run the frond-end load balancer and load generator on separate machines with network links to each of the service hosts.

A. Performance Modeling

Performance models provide a powerful tool for prediction of performance metrics, such as utilization or response time. These predictions enable smart capacity planning, especially in a micro-service environment where containers can be added or removed within seconds. Examples for such performance models are RESOLVE [39], ROBOCOP [40], PCM [1], SAMM [41], CACTOS [42] and UML MARTE [43]. These models have a limited number of real world case studies, making it difficult to evaluate their applicability for real world scenarios. Additionally, quantitative comparison of performance models is challenging, as common case studies do not exist.

We showcase that the TeaStore is well suited as a case study for advanced performance modeling concepts. We use a novel modeling mechanism, modeling the TeaStore. Using the TeaStore, we show that this novel mechanism can improve modeling accuracy when compared to standard state-of-the-art approaches. This, in reverse, highlights the TeaStore's ability to evaluate the applicability and modeling accuracy of novel formalisms. Specifically, we evaluate the modeling concepts for parametric dependencies of Eismann et al. [44] using the TeaStore. This modeling formalism has the capability of modeling and predicting the impact of the changes within a workload profile, which can be mapped to a parameter in the underlying model. We compare this novel formalism to a standard state-of-the-art modeling method, by also constructing a model of the TeaStore using the Descartes Modeling Language (DML) [45]. Both models are used to predict the utilization of previously unseen deployments under different load levels. Using the TeaStore, we aim to show that the ability of the novel modeling formalism to capture parametric changes in the workload profile increases prediction accuracy in comparison to the state-of-the-art method, which does not model this aspect. This comparison shows how the TeaStore can be used as a case study to highlight the benefits of advanced performance modeling concepts.

The category page can display different amounts of products per page, based on user preference. We investigate the question of how changing the default number from five products per page to ten products per page impacts the performance. We assume that for a default of five products per page, some users manually switch to ten or twenty products per page, leading to the distribution shown in Equation 1, where $P_5(x)$ denotes the probability of any given user displaying the amount of x products per page, when a default value of 5 is pre-configured.

$$P_5(x) = \begin{cases} 0.9 & \text{if } x = 5 \\ 0.09 & \text{if } x = 10 \\ 0.01 & \text{if } x = 20 \\ 0 & \text{else.} \end{cases} \quad (1)$$

Equation 2 shows the assumed distribution for a default of ten products per page, where $P_{10}(x)$ denotes the probability of any given user displaying the amount of x products per page, when a default value of 10 is pre-configured. We aim to predict the impact of this change to the products per page on the performance of the TeaStore.

$$P_{10}(x) = \begin{cases} 0 & \text{if } x = 5 \\ 0.99 & \text{if } x = 10 \\ 0.01 & \text{if } x = 20 \\ 0 & \text{else.} \end{cases} \quad (2)$$

We model the software architecture based on the TeaStore architecture shown in Figure 1. To parameterize the service demands in the static model, we deploy each service on a bare-metal server as shown in Figure 4a, measure the utilization for a load of 1000 requests per second, using the usage distribution of Equation 1, and calculate the service demands according to Equation 3. The service demand defines the average time the CPU spends serving one request at the respective service [46].

$$SerD = U/\lambda, \quad (3)$$

where U is the utilization and λ is the arrival rate. For the parametric model, we use linear regression to derive the Equations 4-6 for the service demands of the WebUI, ImageProvider and Persistence services, respectively.

$$SerD_{Web} = 0.0034 + 0.00016 * ProductsPerPage. \quad (4)$$

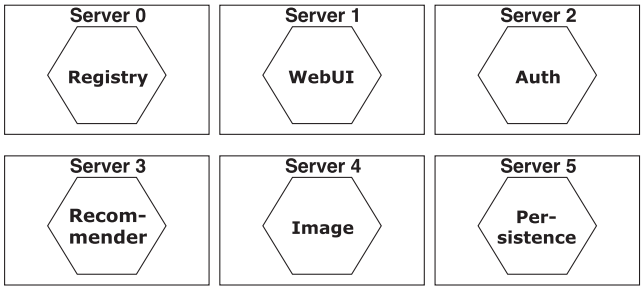
$$SerD_{Img} = 0.0012 + 0.00011 * ProductsPerPage. \quad (5)$$

$$SerD_{Per} = 0.0022 + 0.00005 * ProductsPerPage. \quad (6)$$

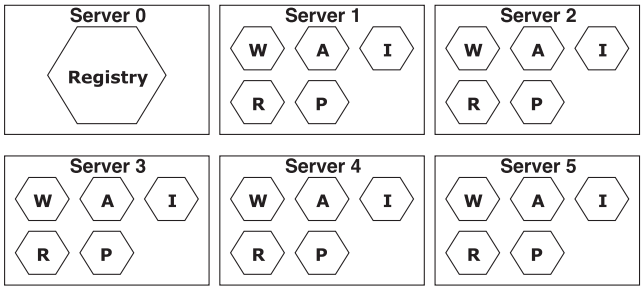
The service demand for the Authentication service remains static, as it is not influenced by the number of products per page. In both models, the recommender service was not included, since no recommendations are displayed on the categories page.

To evaluate the resulting models, we deploy an instance of every service on five servers to represent a production system, as shown in Figure 4b. For each server, we measure the CPU utilization under a load of 1000, 2000, 3000, 4000 and 5000 requests per second for the usage distributions from Equations 1 and 2. We compare the measured utilizations with the predicted utilizations using the static model and the parametric model for both distributions. Figure 5 shows the absolute prediction errors for both models. For a default of five products per page, both the static and the parametric model have an absolute utilization prediction error of $< 5\%$. This is expected, as this is the scenario the static model was build for and calibrated with. However, for a default of ten products per page, the parametric model significantly outperforms the static model. Using the TeaStore as a case study, we are able to highlight the benefit of modeling parametric dependencies. This illustrates the TeaStore's applicability for model evaluation in one specific context.

Aside from parametric dependencies, the TeaStore provides many opportunities to evaluate performance modeling concepts due to its performance properties. In the following, we list a number of challenges, which could be evaluated using the TeaStore. We also list some example approaches that motivate or tackled these issues in the past:



(a) Calibration deployment



(b) Evaluation deployment

Fig. 4: Deployments for model prediction. Services are abbreviated to their first letter.

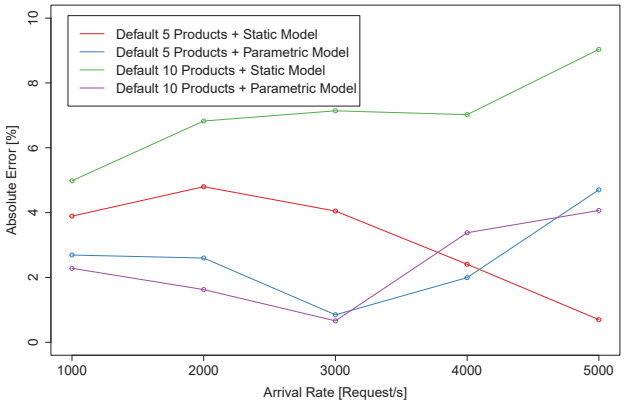


Fig. 5: Absolute prediction error of the static and parametric model for two workload profiles and five workload intensities.

Deployment options The TeaStore offers a large variety of deployment options, as each service can be individually deployed and scaled. The prediction of resource utilizations and response times for previously unseen deployments and configurations is the most common use case for performance engineering. This task overlaps with the remaining challenges and therefore offers itself as a scenario for benchmarking of performance modeling approaches.

Internal state For some services, the performance does not only depend on the request, but also the internal state of the service. In the TeaStore, the database size influences the service demand of the persistence provider service. The number of entries in the database can dynamically change during operation, leading to changing persistence provider service demands. See e.g., [47].

Caching Caching mechanisms are challenging to model, as the behavior of a service with caching depends on its workload profile. The more frequent a small subset of items is requested, the more effective caching becomes. The Persistence and Image Provider of the TeaStore implement caching. The implementation of the image provider cache is known and can be modeled as a white-box, whereas the persistence provider cache is part of the JPA implementation and has to be modeled as a black-box. See e.g., [48], [49]

Network For distributed applications, network delays can influence their response time and can be a bottleneck which limits the maximum throughput. This means, the network topology and limitations need to be taken into account in a performance model to prevent significant prediction errors. As instances of each TeaStore service can be deployed on different machines, potentially even in different data centers, network delays can significantly influence the system response times. The Image Provider transmits potentially large images over the network, which can lead to a network bottleneck. See e.g., [50].

Load types The CPU of a server is the most common bottleneck for applications, however the I/O capacity or the available memory can also be a bottleneck. Additionally, the load profile of two services deployed on the same machine influences how well they run in parallel. The authentication service causes CPU load, the Image provider I/O load and both the Image and Persistence provider caches cause memory load. Therefore, modeling of different load types is necessary to accurately predict the performance of the TeaStore. See e.g., [45].

Startup behavior Some systems dynamically add or remove service instances in order to adapt to changing workload. Whether a service starts instantly or takes some time until it is available, is an important factor when predicting the response times of such systems. The TeaStore services cover both cases. New Recommender instances need to be trained before they can process requests and Image provider instances need to generate the image files upon startup. For the remaining services new instances can be added and removed within seconds. See e.g., [51].

Alternate Implementations Deciding between multiple implementations of the same component or service is a classic performance modeling challenge. The TeaStore provides multiple recommender algorithms and image provider caches, which provides the opportunity for configuration optimization case studies. See e.g., [52].

Timed tasks In most cases, the load of a system depends on the number of requests it receives. Sometimes, a system also regularly performs some tasks without user input, usually some kind of maintenance tasks. Modeling this

behavior explicitly is important, as this load occurs independently of the user behavior. The Recommender service can be configured to be retrained at a regular interval, which is a realistic example of such a maintenance task.

We show that the TeaStore can be used as a case-study for performance modeling approaches. It provides sufficient complexity to challenge existing approaches. The open source nature of the TeaStore enables white-box modeling and any measurements using the TeaStore can be easily replicated, since we provide docker containers with integrated monitoring. Therefore, the TeaStore is a suitable case-study for a quantitative comparison between performance modeling approaches. These comparisons can be done using manually created models or automatically extracted models using approaches such as [53], [54], [55], [56]. Using automatically extracted models would allow comparisons of full tool-chains, provided by the respective modeling formalisms, in a realistic usage scenario.

B. Auto-Scaling

In order to show that the TeaStore works in an elastic manner and can thus be used for auto-scaling experiments, we stress an early development version of the TeaStore using workloads derived from two different real-world traces (FIFA World Cup 1998 [57] and BibSonomy [58]). We employ a common, generic auto-scaler [18] to automatically scale the store at run-time as the load intensity varies. We evaluate the quality of the scaler's decisions using a set of standard auto-scaler evaluation metrics.

1) Workload: We stress the TeaStore using load intensity profiles based on different real-world workloads: (i) BibSonomy and (ii) FIFA World Cup 1998. We select a sub-set lasting four days from each of those traces. The BibSonomy trace represents HTTP requests to the social bookmarking system BibSonomy (see Benz et al. [58]) during April 2017. The FIFA[9] trace is a popular trace that was analyzed by Arlitt and Tai [57]. The FIFA trace represents HTTP requests to the FIFA servers during the world championship between April and June 1998. We modify the traces to cover the load intensity range between a low load that can be covered using a minimal TeaStore deployment and a high load level that reaches the maximum capacity of the potential deployments for this scenario. On the time scale, the experiment is modified so that the load intensity variations of the original four days are executed within one hour.

2) Auto-Scaler: In 2009, Chieu et al. [18] present a reactive scaling algorithm for horizontal scaling. This mechanism provisions Virtual Machines (VMs) based on an application's scaling indicators. Among other things, the indicators consist of the number of active connections or the number of requests per second. The auto-scaler monitors these indicators for each VM and calculates the moving average. Next, the virtual machines with active sessions above or below given thresholds are determined. Finally, if all virtual machines have active sessions above a given threshold, new instances are provisioned. Upon detecting VMs with active sessions below the threshold and with at least one virtual machine that has no active session, idle instances are removed.

[9]FIFA Source: http://ita.ee.lbl.gov/html/contrib/WorldCup.html

TABLE II: Mapping of config. number and used containers.

	Configuration								
Service	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**	**#9**
WebUI	1	1	2	3	3	4	5	5	6
Image Provider	1	1	2	3	3	4	5	5	6
Authentication	1	2	3	4	5	6	7	8	9
Recommender	1	1	1	2	2	2	3	3	3
Persistence	1	2	2	3	4	4	5	6	6

In these experiments we aim to use a representative auto-scaler. Consequently, we choose this reactive technique, as the underlying mechanism is simple and straight-forward. Based on this simplicity, this reactive generic auto-scaling-mechanism can and does serve as baseline approach when comparing state-of-the-art research auto-scalers. For our experiments, we use a re-implementation of this reactive algorithm that scales docker containers and name it *React*. In the evaluation framework of A. Ilyushkin et. al [59], React shows a generic and stable scaling performance.

Scaling distributed applications is still an open research challenge and currently not supported by many state-of-the-art auto-scalers and not by our baseline auto-scaler. To account for this, we provide a best-effort list of which service to scale at which stage. This list acts as a mapping between the distributed services of the TeaStore and the expected single service instances (scaling units) typically addressed by auto-scalers. Specifically, in our case, the auto-scaler under test always deploys a full-stack as the first scaling unit, then one Authentication and one Persistence instance as the second unit and, finally, one WebUI, Authentication, and Image instance each as the third scaling unit. Each service instance is limited to one virtual CPU core on the host machine. For additional scaling units, these three steps are repeated on the next available physical host (see Table II). We use up to three physical server, resulting in a maximum configuration with a total of 30 service containers.

3) Quantifying Scaling Behaviour: In order to evaluate the scaling decisions made by React, we consider the proportion of failed transactions, the average response time and a set of system-oriented elasticity metrics endorsed by the Research Group of the Standard Performance Evaluation Corporation (SPEC) [60]. In particular, we focus on the *wrong provisioning time share*.

The wrong provisioning time share captures the time in which the system is in an under-provisioned (or over-provisioned) state during the experiment interval. That is, the *under-provisioning time share* τ_U describes the time relative to the measurement duration, in which the system is under-provisioned. Similarly, the *over-provisioning time share* τ_O describes the time relative to the measurement duration in which the system is in an over-provisioned state. The range of this metric is the interval $[0, 100]$. The best value of 0 is achieved, when the system features no over- or under-

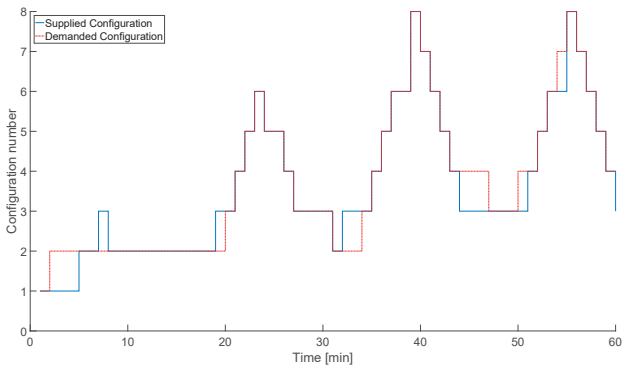


Fig. 6: Scaling behavior for the FIFA trace.

provisioning during the measurement. We define both metrics τ_U and τ_O as follows:

$$\tau_U[\%] := \frac{100}{T} \cdot \sum_{t=1}^{T} max(sgn(d_t - s_t), 0)\Delta t$$

$$\tau_O[\%] := \frac{100}{T} \cdot \sum_{t=1}^{T} max(sgn(s_t - d_t), 0)\Delta t \text{ , where}$$

d_t is the minimal amount of scaling units required to meet the Service Level Objectives (SLOs) under the load intensity at time t, s_t is the resource supply at time t, and T is the experiment duration. Δt denotes the time interval between the last and the current change either in demand d or supply s.

To know whether or not the system is in an under-provisioned or over-provisioned state, we execute separate off-line calibration measurements, which measure the average throughput of each target configuration in an over-loaded state. Based on this configuration, capacity information, and the load intensity of our traces, we are able to derive the required resource configuration for each point in time, which we can then use for calculating the time share metrics.

4) Evaluation: The scaling behavior of React on both the FIFA and BibSonomy traces is shown in Figure 6 and in Figure 7. Both figures are structured as follows: The

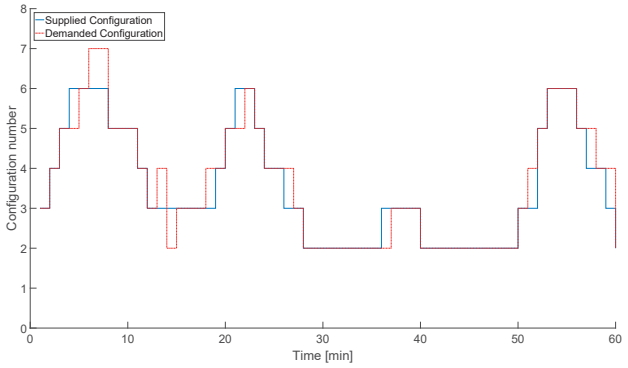


Fig. 7: Scaling behavior for the BibSonomy trace.

TABLE III: Result metric overview for both traces.

Metric	Fifa	BibSonomy
τ_U under-provisioning time share	15%	13%
τ_O over-provisioning time share	7%	6%
Proportion of failed transactions	1%	1%
Average response time	0.18 s	0.15 s

horizontal axis shows the experiment time in minutes; the vertical axis represents the current number of scaling units (the configuration). Table II shows the mapping between the number and used containers. The red, dashed curve represents the required configuration, which has been derived using the separate off-line calibration tests, and the blue curve shows the supplied configuration, as measured during the experiment. For both traces, React scales the system in a similar fashion. That is, in both traces, the supplied configuration matches the required configuration for long periods. However, some deviations between supplied and required configurations occur. In Figure 6, for example, the system is in an under-provisioned state in the entire interval between minute 2 and 5. Overall, the under-provisioning and over-provisioning time-shares are equal or below 15% in both traces (see Table III), indicating good scaling behavior. In addition to the low time-share metrics, the assertion of good scaling behavior is backed up by the observation that the proportion of failed transactions is below 2% and the average response time is lower than 0.19 s for both traces.

The results of our auto-scaler tests indicate that the TeaStore can be used to compare the elasticity and performance of state-of-the art auto-scalers. Specifically, they show that the TeaStore can even be used for comparing commonly used and state-of-the art autoscalers, even though these scalers are usually limited to single tier scaling. In addition, the results show that the TeaStore exhibits robust behavior during run-time scaling, as the proportion of failed transactions is below 2%. This characteristic is achieved through the micro-service architecture and the client-side load balancers. The results also indicate that the TeaStore is sufficiently scalable to allow for experiments of this kind.

Many open challenges remain, despite the good performance of the *React* auto-scaler. In these experiments, the service deployment order was fixed and the auto-scalers do not have to decide which service to place on which machine, but rather when to add or remove the next configuration from the pre-defined list of configurations. Distributed application deployment decisions of this kind remain an open challenge. Furthermore, load profiles may be more complex, as the used profile does not change the database and does not change in the user actions performed over time. Real-world auto-scalers face the challenge of evolving user-behavior and changes in request service demands over time.

C. Energy-Efficiency Analysis

Energy efficiency and power prediction methods, such as [5], [6] are often employed to solve a placement problem for services in distributed systems. The underlying challenge is

that different distributions of application services across physical hosts may not only result in different performance behavior but also in differences in overall power consumption. This section demonstrates this effect using the TeaStore. We show that different distributions of the TeaStore's services can result in different performance and in different power consumption both on homogeneous and heterogeneous systems.

For these experiments, we use an increasing load intensity profile. The load profile starts at 8 requests per second and increases to 2000 requests per second over the time of four minutes. Request content is, again, specified using the user browse profile for the 128 users accessing the store. Depending on the current SUT configuration, some of the four minutes are spent in an under-provisioned state, in which the load intensity exceeds the capacity of the SUT. We measure the power consumption of the physical servers and throughput of the TeaStore during the entire run. However, we only take those measurements into account, which are measured during the time in which load arrives at the system. Each measurement is taken on a per-second basis and thus tightly coupled to the current load intensity.

We calculate the following metrics based on the throughput and power consumption:

1) **Energy Efficiency**: In accordance to the SPEC Power methodology [61], we define energy efficiency as the ratio of throughput to power consumption:

$$Efficiency[J^{-1}] = \frac{Throughput[s^{-1}]}{Power[W]}$$

   As energy efficiency is a ratio, we aggregate multiple energy efficiency scores using the geometric mean.
2) **Estimated Capacity**: We estimate the throughput capacity of each configuration by averaging the last 50 seconds of our load profile. Note that all configurations are operating at maximum load (capacity) at this time.
3) **Maximum Power Consumption**: The maximum power consumption measured (in Watts). It indicates the power load that the configuration can put on the SUT.

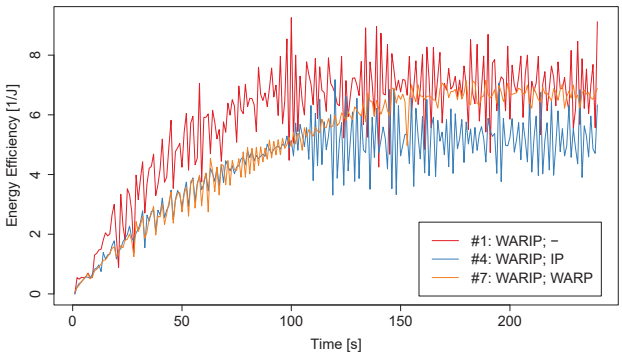


Fig. 8: Energy efficiency for linearly increasing load. Services are abbreviated to their first letter.

TABLE IV: Energy Efficiency on Homogeneous Servers. Services are abbreviated to their first letter.

#	Serv.1	Serv.2	Capacity	Max Pwr	Eff.
1	WARIP	-	779.7 ±[29.7]	**114.4 W**	**5.3**
2	WI	ARP	1177.5 ±[31.5]	193.6 W	4.2
3	WAI	RP	883.4 ±[39.4]	175.8 W	3.8
4	WARIP	IP	863.0 ±[40.5]	173.5 W	3.9
5	WARIP	AIP	1228.7 ±[18.9]	208.4 W	4.2
6	WAIP	WARP	1231.8 ±[18.7]	203.7 W	4.3
7	WARIP	WAIP	1404.1 ±[14.5]	217.9 W	4.3
8	WARIP	WARIP	**1413.2** ±[14.7]	217.7 W	4.3

1) Energy Efficiency on Homogeneous Systems: We run the TeaStore with un-restrained docker containers on up to two of our testing servers. Table IV shows the estimated capacity, maximum power, and geomean energy efficiency for different TeaStore deployments on those servers (Service names in the table are abbreviated to their first letter). The table confirms our previous assertion that the store performs differently depending on the service distribution. Capacity (maximum throughput) varies significantly for the different deployments, with some two-server deployments barely exceeding the capacity of the single server deployment and others almost doubling it.

The single server deployment (deployment #1) features the lowest performance, but also the lowest power consumption. As a result, it has the highest energy efficiency among all tested configurations. This is mostly due to our increasing stress test profile. At low load, as the load increases, the single system is still capable of handling all requests, while also consuming less power. At high load, it operates at capacity, but still consumes less power than the two-server setups. Figure 8 visualizes the energy efficiency over time for the single-server and two selected two-server deployments. The figure shows that an efficient two-server deployment can reach a similar energy efficiency as the single-server deployment at maximum load. However, some low performance deployments are incapable of reaching this efficiency and are overall less efficient due to the power overhead of the second server.

Among the two-server deployments, maximum power consumption is usually greater for those deployments with greater capacity, but some notable differences exist, which indicate room for power and efficiency optimization, even on a homogeneous system. Two notable examples emerge: Comparing deployment #2 with deployment #3, shows that deployment #2, which deploys the WebUI and Image services on one and the Auth, Recommender, and Persistence services on the other server has both a better performance, as well as smaller power footprint than deployment #3, which deploys the WebUI, Auth, and Image services on one server and the Image and Persistence services on another server. Consequently, deployment #2 features a better energy efficiency. In this example, one deployment is obviously better than another and the TeaStore could be used to evaluate if a prediction or management mechanism actually selects the better option. However, in some cases power does not scale the same as performance. This case is hinted at when comparing deployment #5 and #6.

TABLE V: Energy Efficiency on Heterogeneous Servers. Services are abbreviated to their first letter.

#	8 core	4 core	Capacity	Max Pwr	Eff.
1	WARIP	-	779.7 ±[29.7]	114.4 W	5.3
2	AIP	WARIP	781.1 ±[11.1]	**163.1** W	3.9
3	WARIP	AIP	**1207.3** ±[23.4]	189.5 W	**4.6**
4	WAIP	WARIP	1011.9 ±[24.7]	179.6 W	4.4
5	WARIP	WAIP	1067.7 ±[26.7]	187.0 W	4.3
6	WARIP	WARIP	1003.9 ±[24.9]	179.7 W	4.1

Both deliver equal performance, but deployment #6 consumes slightly less power and is therefore a bit more efficient.

2) Energy Efficiency on Heterogeneous Systems: For our measurements on heterogeneous systems we replace the second server with an HP ProLiant DL20 system, which features an Intel Xeon E3-1230 v5 processor with 4 cores at 3.5 GHz and 16 GB RAM. This second server does not offer as much performance and consumes less power compared to its 8 core counterpart. Naturally, when deploying on this heterogeneous system, the order of deployment matters, as servers differ in power and performance.

Table V shows the measurement results of the heterogeneous system. It shows the performance, power, and energy efficiency for selected deployments. It illustrates the effect that deployment order has on the heterogeneous system, especially regarding deployments #2 and #3, which are the same deployment, except that they deploy each respective stack on the different server. Deployment #2 deploys the full stack on the smaller server and replicates some components on the larger machine, wheres deployment #3 does the reverse. Although deployment #3 consumes more power than #2, it has a far better performance and greater overall efficiency. It should also be noted that deployments with fewer services on the smaller machine seem to be more efficient in the heterogeneous environment compared to the respective deployments in the homogeneous environment, which deploy the smaller stack on an equivalent machine. Deployment #5 corresponds to deployment #7 on the homogeneous system (see Table IV), which is the most efficient system in that context. However, on the heterogeneous system, it is trumped in performance and efficiency by deployment #3, which places fewer services on the smaller machine.

In addition, the heterogeneous system demonstrates an efficiency – performance trade-off when compared to the homogeneous system. The most efficient heterogeneous deployment has a slightly lower performance capacity than the best homogeneous one, yet consumes less power and has a better energy efficiency.

Overall, our experiments show that the TeaStore exhibits different performance and power behavior depending on deployment, both on heterogeneous and homogeneous systems. Due to this, it can be used to evaluate the prediction accuracy of power prediction mechanisms. In addition, some of our configurations feature a performance – efficiency trade-off, which is highly relevant for power and performance management, showing that the TeaStore can be used to evaluate such management approaches.

V. Conclusion

This paper introduces the TeaStore, a test and reference application intended to serve as a benchmarking framework for researchers evaluating their work[10]. The TeaStore is designed to offer the degrees of freedom and performance characteristics required by software management, prediction, and analysis research. Specifically, the TeaStore is designed to be used in one of three target domains: Evaluation of software performance models and model-extractors, evaluation of run-time software management methods, such as auto-scalers, and evaluation of software energy-efficiency, power models and optimization methods.

The TeaStore is a distributed micro-service-based application, consisting of five separate services, each of which can be replicated, added, and removed at run-time. The TeaStore's services are available as Docker containers and as manually deployable applications. It deviates from existing testing and benchmarking applications through its focus on the target research application scenarios. This focus has influenced the design, implementation, and performance characteristics of the store. The TeaStore thus also comes with a pre-instrumented variant that collects performance data at run-time, further enhancing its use in the intended contexts.

We demonstrate the TeaStore's use for the target scenarios using separate use-case experiments:

1) We create and calibrate software performance models for the TeaStore. We use these models to predict the store's utilization for different usage profiles in a distributed setting. With these experiments, we show that the TeaStore can be used to evaluate the benefit and modeling accuracy of novel performance modeling formalisms.
2) We run the TeaStore in an elastic environment using a state-of-the-art auto-scaler, demonstrating its run time scalability. This experiment also highlights the limitations of current auto-scalers regarding micro-service applications and multiple different services in general.
3) We analyze the energy efficiency of different deployments for the TeaStore, showing the non-trivial power and performance effects that placement decisions can have. In addition, we show that some store configurations offer a trade-off between energy efficiency and performance, which can be employed by management mechanisms.

Researchers may decide to use the TeaStore and the corresponding testing tools and profiles in any of these three primary application scenarios. The TeaStore can also be used in additional settings to achieve evaluation results that demonstrate the applicability of their work, while also enhancing comparability of their results.

[10]TeaStore setup: https://github.com/DescartesResearch/TeaStore/wiki/Testing-and-Benchmarking

REFERENCES

[1] S. Becker, H. Koziolek, and R. Reussner, "The palladio component model for model-driven performance prediction," *Journal of Systems and Software*, vol. 82, no. 1, pp. 3–22, 2009.

[2] A. Ilyushkin, A. Ali-Eldin, N. Herbst, A. V. Papadopoulos, B. Ghit, D. Epema, and A. Iosup, "An Experimental Performance Evaluation of Autoscaling Policies for Complex Workflows," in *Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering (ICPE 2017)*. New York, NY, USA: ACM, April 2017, best Paper Candidate (1/4).

[3] I. Lee and R. K. Iyer, "Software dependability in the tandem guardian system," *IEEE Transactions on Software Engineering*, vol. 21, no. 5, pp. 455–467, 1995.

[4] B. Littlewood and L. Strigini, "Validation of ultra-high dependability for software-based systems," in *Predictably Dependable Computing Systems*. Springer, 1995, pp. 473–493.

[5] A. Beloglazov, J. Abawajy, and R. Buyya, "Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing," *Future Gener. Comput. Syst.*, vol. 28, no. 5, pp. 755–768, May 2012. [Online]. Available: http://dx.doi.org/10.1016/j.future.2011.04.017

[6] R. Basmadjian, N. Ali, F. Niedermeier, H. de Meer, and G. Giuliani, "A Methodology to Predict the Power Consumption of Servers in Data Centres," in *Proceedings of the 2Nd International Conference on Energy-Efficient Computing and Networking*, ser. e-Energy '11. New York, NY, USA: ACM, 2011, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/2318716.2318718

[7] J. von Kistowski, J. Beckett, K.-D. Lange, H. Block, J. A. Arnold, and S. Kounev, "Energy Efficiency of Hierarchical Server Load Distribution Strategies," in *Proceedings of the IEEE 23nd International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2015)*. IEEE, October 2015.

[8] C. M. Aderaldo, N. C. Mendonça, C. Pahl, and P. Jamshidi, "Benchmark requirements for microservices architecture research," in *Proceedings of the 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering*. IEEE Press, 2017, pp. 8–13.

[9] J. Happe, H. Koziolek, and R. Reussner, "Facilitating performance predictions using software components," *IEEE Software*, vol. 28, no. 3, pp. 27–33, 2011.

[10] F. Willnecker, M. Dlugi, A. Brunnert, S. Spinner, S. Kounev, and H. Krcmar, "Comparing the Accuracy of Resource Demand Measurement and Estimation Techniques," in *Computer Performance Engineering - Proceedings of the 12th European Workshop (EPEW 2015)*, ser. Lecture Notes in Computer Science, M. Beltrán, W. Knottenbelt, and J. Bradley, Eds., vol. 9272. Springer, August 2015, pp. 115–129.

[11] *RUBiS User's Manual*, May 2008.

[12] D. Inc., "Dell DVD Store," https://linux.dell.com/dvdstore/, 2011, Accessed: 13.10.2017.

[13] Weaveworks Inc., "Sock Shop: A Microservice Demo Application," https://github.com/microservices-demo/microservices-demo, 2017, Accessed: 19.10.2017.

[14] A. Brunnert, A. van Hoorn, F. Willnecker, A. Danciu, W. Hasselbring, C. Heger, N. Herbst, P. Jamshidi, R. Jung, J. von Kistowski, A. Koziolek, J. Kross, S. Spinner, C. Vögele, J. Walter, and A. Wert, "Performance-oriented DevOps: A research agenda," SPEC Research Group — DevOps Performance Working Group, Standard Performance Evaluation Corporation (SPEC), Tech. Rep. SPEC-RG-2015-01, August 2015. [Online]. Available: https://research.spec.org/fileadmin/user_upload/documents/wg_devops/endorsed_publications/SPEC-RG-2015-001-DevOpsPerformanceResearchAgenda.pdf

[15] Standard Performance Evaluation Corporation (SPEC), "SPEC jEnterprise 2010 Design Document," https://www.spec.org/jEnterprise2010/docs/DesignDocumentation.html, May 2010, Accessed: 16.10.2017.

[16] P. Ezhilchelvan and I. Mitrani, "Optimal provision of multiple service types," in *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2016 IEEE 24th International Symposium on*. IEEE, 2016, pp. 21–29.

[17] Z. Gong, X. Gu, and J. Wilkes, "Press: Predictive elastic resource scaling for cloud systems," in *Network and Service Management (CNSM), 2010 International Conference on*. Ieee, 2010, pp. 9–16.

[18] T. C. Chieu, A. Mohindra, A. A. Karve, and A. Segal, "Dynamic scaling of web applications in a virtualized cloud computing environment," in *E-Business Engineering, 2009. ICEBE'09. IEEE International Conference on*. IEEE, 2009, pp. 281–286.

[19] IBM, "ACME Air," https://github.com/acmeair/acmeair, 2015, Accessed: 19.10.2017.

[20] K. Bastani, "Spring Cloud Example Project," https://github.com/kbastani/spring-cloud-microservice-example, 2015, Accessed: 19.10.2017.

[21] .NET Foundation, "MusicStore (test application)," https://github.com/aspnet/MusicStore, 2017, Accessed: 18.10.2017.

[22] E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite, and W. Zwaenepoel, "Performance comparison of middleware architectures for generating dynamic web content," in *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*. Springer-Verlag New York, Inc., 2003, pp. 242–261.

[23] E. Cecchet, J. Marguerite, and W. Zwaenepoel, "Performance and scalability of ejb applications," in *ACM Sigplan Notices*, vol. 37, no. 11. ACM, 2002, pp. 246–261.

[24] O. Consortium, "Rice University Bidding System (RUBiS)," http://rubis.ow2.org/index.html, 2009, Accessed: 13.10.2017.

[25] CloudScale Consortium, "CloudStore," https://github.com/CloudScale-Project/CloudStore, 2016, Accessed: 18.10.2017.

[26] S. Lehrig, R. Sanders, G. Brataas, M. Cecowski, S. Ivanšek, and J. Polutnik, "Cloudstoretowards scalability, elasticity, and efficiency benchmarking and analysis in cloud computing," *Future Generation Computer Systems*, vol. 78, pp. 115–126, 2018.

[27] G. Brataas, N. Herbst, S. Ivansek, and J. Polutnik, "Scalability Analysis of Cloud Software Services," in *Companion Proceedings of the 14th IEEE International Conference on Autonomic Computing (ICAC 2017), Self Organizing Self Managing Clouds Workshop (SOSeMC 2017)*. IEEE, July 2017.

[28] A. Rausch, R. Reussner, R. Mirandola, and F. Plasil, "The common component modeling example," *Lecture notes in computer science*, vol. 5153, 2008.

[29] Oracle and S. Microsystems, "JPetStore 2.0," http://www.oracle.com/technetwork/java/index-136650.html, 2005, Accessed: 17.10.2017.

[30] P. Software, "Spring PetClinic," https://github.com/spring-projects/spring-petclinic, 2016, Accessed: 19.10.2017.

[31] A. van Hoorn, J. Waller, and W. Hasselbring, "Kieker: A framework for application performance monitoring and dynamic software analysis," in *Proceedings of the 3rd joint ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*. ACM, April 2012, pp. 247–248. [Online]. Available: http://eprints.uni-kiel.de/14418/

[32] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst, "Continuous monitoring of software services: Design and application of the kieker framework," Kiel University, Forschungsbericht, November 2009. [Online]. Available: http://eprints.uni-kiel.de/14459/

[33] D. Lemire and A. Maclachlan, "Slope one predictors for online rating-based collaborative filtering," in *Proceedings of the 2005 SIAM International Conference on Data Mining*. SIAM, 2005, pp. 471–475.

[34] S. Spinner, G. Casale, X. Zhu, and S. Kounev, "LibReDE: A Library for Resource Demand Estimation," in *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014)*. New York, NY, USA: ACM Press, March 2014, pp. 227–228. [Online]. Available: http://doi.acm.org/10.1145/2568088.2576093

[35] D. Okanović, A. van Hoorn, C. Heger, A. Wert, and S. Siegl, "Towards performance tooling interoperability: An open format for representing execution traces," in *Proceedings of the 13th European Workshop on Performance Engineering (EPEW '16)*. Springer, 2016.

[36] J. von Kistowski, M. Deffner, and S. Kounev, "Run-time Prediction of Power Consumption for Component Deployments," in *Proceedings of the 15th IEEE International Conference on Autonomic Computing (ICAC 2018)*, September 2018.

[37] J. von Kistowski, N. Herbst, S. Kounev, H. Groenda, C. Stier, and S. Lehrig, "Modeling and Extracting Load Intensity Profiles," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 11, no. 4, pp. 23:1–23:28, January 2017. [Online]. Available: http://doi.acm.org/10.1145/3019596

[38] B. Schroeder, A. Wierman, and M. Harchol-Balter, "Open versus closed: a cautionary tale," in *Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3*, ser. NSDI'06. Berkeley, CA, USA: USENIX Association, 2006, pp. 18–18. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267680.1267698

[39] M. Sitaraman, G. Kulczycki, J. Krone, W. F. Ogden, and A. L. N. Reddy, "Performance specification of software components," *SIGSOFT Softw. Eng. Notes*, vol. 26, no. 3, pp. 3–10, May 2001. [Online]. Available: http://doi.acm.org/10.1145/379377.375223

[40] E. Bondarev, P. de With, M. Chaudron, and J. Muskens, "Modelling of input-parameter dependency for performance predictions of component-based embedded systems," in *31st EUROMICRO Conference on Software Engineering and Advanced Applications*. Vienna, Austria: EUROMICRO, Aug 2005, pp. 36–43.

[41] S. Becker, L. Bulej, T. Bures, P. Hnetynka, L. Kapova, J. Kofron, H. Koziolek, J. Kraft, R. Mirandola, J. Stammel, G. Tamburelli, and M. Trifu, "Q-impress consortium," www.q-impress.eu/wordpress/wp-content/uploads/2009/05/d21-service_architecture_meta-model.pdf, 2017, accessed: 2017.03.16.

[42] H. Groenda, C. Stier, J. Krzywda, J. Byrne, S. Svorobej, G. G. Castañé, Z. Papazachos, C. Sheridan, D. Whigham, C. Hauser *et al.*, "Cactos toolkit version 2: accompanying document for prototype deliverable d5. 2.2," 2017.

[43] S. Gérard and B. Selic, "The uml–marte standardized profile," *IFAC Proceedings Volumes*, vol. 41, no. 2, pp. 6909–6913, 2008.

[44] S. Eismann, J. Walter, J. von Kistowski, and S. Kounev, "Modeling of Parametric Dependencies for Performance Prediction of Component-based Software Systems at Run-time," in *2018 IEEE International Conference on Software Architecture (ICSA)*, May 2018.

[45] N. Huber, F. Brosig, S. Spinner, S. Kounev, and M. Bähr, "Model-Based Self-Aware Performance and Resource Management Using the Descartes Modeling Language," *IEEE Transactions on Software Engineering (TSE)*, vol. 43, no. 5, 2017. [Online]. Available: http://dx.doi.org/10.1109/TSE.2016.2613863

[46] S. Spinner, G. Casale, F. Brosig, and S. Kounev, "Evaluating Approaches to Resource Demand Estimation," *Performance Evaluation*, vol. 92, pp. 51 – 71, October 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0166531615000711

[47] L. Happe, B. Buhnova, and R. Reussner, "Stateful component-based performance models," *Softw. Syst. Model.*, vol. 13, no. 4, pp. 1319–1343, Oct. 2014.

[48] G. Bianchi, A. Detti, A. Caponi, and N. Blefari Melazzi, "Check before storing: What is the performance price of content integrity verification in lru caching?" *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 3, pp. 59–67, Jul. 2013.

[49] M. Garetto, E. Leonardi, and V. Martina, "A unified approach to the performance analysis of caching systems," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, vol. 1, no. 3, p. 12, 2016.

[50] P. Rygielski, M. Seliuchenko, and S. Kounev, "Modeling and Prediction of Software-Defined Networks Performance using Queueing Petri Nets," in *Proceedings of the Ninth International Conference on Simulation Tools and Techniques (SIMUTools 2016)*, 2016, pp. 66–75.

[51] A. V. Papadopoulos, A. Ali-Eldin, K.-E. en, J. Tordsson, and E. Elmroth, "Peas: A performance evaluation framework for auto-scaling strategies in cloud applications," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 1, no. 4, pp. 15:1–15:31, Aug. 2016.

[52] A. Koziolek, D. Ardagna, and R. Mirandola, "Hybrid multi-attribute QoS optimization in component based software systems," *Journal of Systems and Software*, vol. 86, no. 10, pp. 2542 – 2558, 2013.

[53] K. Krogmann, M. Kuperberg, and R. Reussner, "Using genetic search for reverse engineering of parametric behavior models for performance prediction," *IEEE Transactions on Software Engineering*, vol. 36, no. 6, pp. 865–877, 2010.

[54] J. Walter, A. D. Marco, S. Spinner, P. Inverardi, and S. Kounev, "Online Learning of Run-time Models for Performance and Resource Management in Data Centers," in *Self-Aware Computing Systems*, S. Kounev, J. O. Kephart, A. Milenkoski, and X. Zhu, Eds. Berlin Heidelberg, Germany: Springer Verlag, 2017.

[55] F. Willnecker, A. Brunnert, W. Gottesheim, and H. Krcmar, "Using dynatrace monitoring data for generating performance models of java ee applications," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ACM, 2015, pp. 103–104.

[56] J. Walter, C. Stier, H. Koziolek, and S. Kounev, "An Expandable Extraction Framework for Architectural Performance Models," in *Proceedings of the 3rd International Workshop on Quality-Aware DevOps (QUDOS'17)*. ACM, April 2017.

[57] M. Arlitt and T. Jin, "A Workload Characterization Study of the 1998 World Cup Web Site," *IEEE Network*, vol. 14, no. 3, pp. 30–37, 2000.

[58] D. Benz, A. Hotho, R. Jäschke, and more, "The social bookmark and publication management system bibsonomy," *VLDB*, vol. 19, no. 6, pp. 849–875, 2010.

[59] A. Ilyushkin, A. Ali-Eldin, N. Herbst, A. Bauer, A. V. Papadopoulos, D. Epema, and A. Iosup, "An Experimental Performance Evaluation of Autoscalers for Complex Workflows," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (ToMPECS)*, vol. 3, no. 2, pp. 8:1–8:32, April 2018.

[60] N. Herbst and more, "Ready for Rain? A View from SPEC Research on the Future of Cloud Metrics," *CoRR*, vol. abs/1604.03470, 2016.

[61] S. P. E. C. (SPEC), "Power and Performance Benchmark Methodology," November 2012, http://spec.org/power/docs/SPEC-Power_and_Performance_Methodology.pdf.