SuanMing: Explainable Prediction of Performance Degradations in Microservice Applications

Johannes Grohmann
University of Würzburg
Würzburg, Germany
johannes.grohmann@uni-wuerzburg.de

Martin Straesser
University of Würzburg
Würzburg, Germany
martin.straesser@uni-wuerzburg.de

Avi Chalbani
Huawei Technologies
Tel Aviv, Israel
avi.chalbani@huawei.com

Simon Eismann
University of Würzburg
Würzburg, Germany
simon.eismann@uni-wuerzburg.de

Yair Arian
Huawei Technologies
Tel Aviv, Israel
yair.arian@huawei.com

Nikolas Herbst
University of Würzburg
Würzburg, Germany
nikolas.herbst@uni-wuerzburg.de

Noam Peretz
Huawei Technologies
Tel Aviv, Israel
noam.peretz@huawei.com

Samuel Kounev
University of Würzburg
Würzburg, Germany
samuel.kounev@uni-wuerzburg.de

ABSTRACT

Application performance management (APM) tools are useful to observe the performance properties of an application during production. However, APM is normally purely reactive, that is, it can only report about current or past performance degradation. Although some approaches capable of predictive application monitoring have been proposed, they can only report a predicted degradation but cannot explain its root-cause, making it hard to prevent the expected degradation.

In this paper, we present SuanMing—a framework for predicting performance degradation of microservice applications running in cloud environments. SuanMing is able to predict future root causes for anticipated performance degradations and therefore aims at preventing performance degradations before they actually occur. We evaluate SuanMing on two realistic microservice applications, TeaStore and TrainTicket, and we show that our approach is able to predict and pinpoint performance degradations with an accuracy of over 90%.

CCS CONCEPTS

• **Software and its engineering** → **Software performance**; • **Computing methodologies** → *Machine learning approaches*; • **Computer systems organization** → **Cloud computing**.

KEYWORDS

explainability, performance prediction, forecasting, microservices

ICPE '21, April 19–23, 2021, Virtual Event, France

ACM ISBN 978-1-4503-8194-9/21/04…$15.00
https://doi.org/10.1145/3427921.3450248

ACM Reference Format:
Johannes Grohmann, Martin Straesser, Avi Chalbani, Simon Eismann, Yair Arian, Nikolas Herbst, Noam Peretz, and Samuel Kounev. 2021. SuanMing: Explainable Prediction of Performance Degradations in Microservice Applications. In *Proceedings of the 2021 ACM/SPEC International Conference on Performance Engineering (ICPE '21), April 19–23, 2021, Virtual Event, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3427921.3450248

1 INTRODUCTION

Microservice applications [18, 21] are increasingly seen as the main architectural paradigm for developing medium- and large-scale cloud applications [7]. While a microservice architecture offers clear advantages for developing and operating an application, the increase in the number of individual components also increases the perceived complexity of the respective system [11]. Therefore, operators and performance engineers increasingly rely on application performance management (APM) tools to supervise the operation of an application [9, 18]. APM tools and services, like Jaeger[1], Zipkin[2], or Pinpoint[3], allow the collection, processing, and analysis of performance metrics of cloud-based applications.

In general, such tools are limited to reactive performance management, that is, performance degradations can only be detected and addressed after they have occurred in the system. This leads to unavoidable quality of service degradation, negatively impacting the user experience and revenue [12]. Therefore, we envision a proactive APM tool capable of predicting performance degradations before they actually occur.

Current approaches require either low-level hardware measurements [12, 15] or application logs [5, 6, 16, 23–26] in order to deliver their predictions, both of which cannot be assumed to be commonly available using low-overhead cloud APM tools. Additionally, a recent survey of approaches for failure prediction in online environments [13] found that no approach is able to provide a reliable

[1] https://www.jaegertracing.io/
[2] https://zipkin.io/
[3] https://naver.github.io/pinpoint/

explanation for the predicted performance degradation. However, without an explanation indicating the root cause of a predicted performance degradation, its mitigation is challenging.

In this work, we introduce SuanMing, an approach for enabling explainable performance predictions for microservice applications running in cloud environments. SuanMing utilizes tracing data commonly available in APM tools to learn three different models: (i) a forecasting model predicting the user behavior; (ii) a propagation model inferring the behavior of each user request in an application; and (iii) a performance prediction model predicting the performance of services and back-propagating its effect on other dependent services. Based on the model predictions, SuanMing is able to forecast the future state of the application and provide an explanation by pinpointing the respective root-cause service.

The contribution of this paper is two-fold:

- We introduce SuanMing, an approach for predicting performance degradation of microservice applications based on the propagation of internal requests and the back-propagation of service performance, enabling the pinpointing of a root-cause service.
- We present an abstract formalization of the involved prediction tasks in order to enable a modular architecture of the proposed approach.

Operators can use SuanMing as a plugin to their already configured and running monitoring stack to augment the reactive capabilities of their APM tools with a predictive and proactive component that is able to determine—and consequentially avoid—performance degradations before they actually occur. Due to the modular and highly configurable approach of the framework, operators can fine-tune the sensitivity of the approach based on their specific needs. In contrast to related work, SuanMing requires no additional application data for delivering predictions; all information is extracted from the APM tool, and models are continuously updated.

In order to show the benefits of SuanMing, we conduct evaluations on two representative microservice applications. We analyze its performance on the TrainTicket [35] and the TeaStore [31] application using both Pinpoint[3] and a real-world cloud monitoring environment running in the Huawei Cloud[4].

The remainder of this paper is structured as follows: We introduce the SuanMing framework in Section 2 and evaluate its performance in Section 3. We discuss the limitations and future work in Section 4 and summarize related work in Section 5. We conclude the paper in Section 6.

2 APPROACH

Our approach is based on two key observations in microservice architectures. First, we assume that the performance of micro-services is only dependent on the type and amount of requests arriving at each particular service instance. This is based on the assumption that microservice designs should be mostly stateless [7, 31], and therefore all state information is transmitted using the request itself.

Second, a service usually has a limited set of responsibilities [18, 21]. Subsequently, a single user request usually causes a sequence of internal requests to other micro-services in order to realize complex application behaviors. This enables us to split the prediction of large and complex applications into multiple smaller tasks that are individually solvable as the small-scoped services have a predictable performance behavior. The propagation of requests and its resulting performance are also predictable by tracing request call trees. Our idea is a divide-and-conquer approach. We first predict the user requests and their propagation through the application. Then, we use the fine-granular analysis of the individual services to infer the whole application performance and additionally pinpoint the location of the performance degradation without the need for in-depth monitoring data.

In the following, the term *service* represents an application component, which is stateless and has a clear-scoped functionality, i.e., a micro-service. A service always consists of one or multiple *endpoints* . An endpoint is an interface (or workload class) of a service, which can be called by users or other services, e.g., REST endpoints. *User requests* are requests that enter the system from outside the monitored environment. On the contrary, standard *requests* are calls that were issued by other entities (i.e., other services) from inside the system. A *backend service* is a service, which does not issue requests to services and responds to incoming requests only, a *frontend service* is a service responding to user requests.

2.1 Overview

Figure 1 presents our reference architecture containing the different components of SuanMing. The structure enables us to separate the learning of the models and the actual prediction into parallel executable online processes. Additionally, due to the modular structure, it is possible to modify and exchange individual algorithms of the framework without affecting the performance of the other components.

The *Controller* serves as a central synchronization component, responsible for updating times, configurations, and activities. Next, the *Provider* collects and parses incoming data, and stores it into a uniform format using the data storage component. Subsequently, the gathered monitoring information is fed into the *Propagation Trainer* and *Performance Trainer* components. Both modules train a prediction model and store it in the model storage. In contrast, the *Load Forecaster* directly produces a forecast of the expected number of incoming user requests, which is forwarded to the *Predictor*. The Predictor is the central component responsible for predicting the performance of each individual service using the load forecast together with the trained propagation and performance models. Finally, the *Analyzer* compares the predictions with the user-given goals in order to alert and pinpoint any anticipated performance degradation. In the following, we will explain each component in more detail.

2.2 Provider

SuanMing requires two types of training data. First, we need information about the chain of internal requests issued to process an incoming user request. This is required in order to extract the application architecture as well as forecast the number of requests arriving at every service using the propagation model.

Second, performance metrics for every service must be available in order to train the performance models and to do predictions on these monitored values. Depending on the availability, more

[4]https://www.huaweicloud.com/

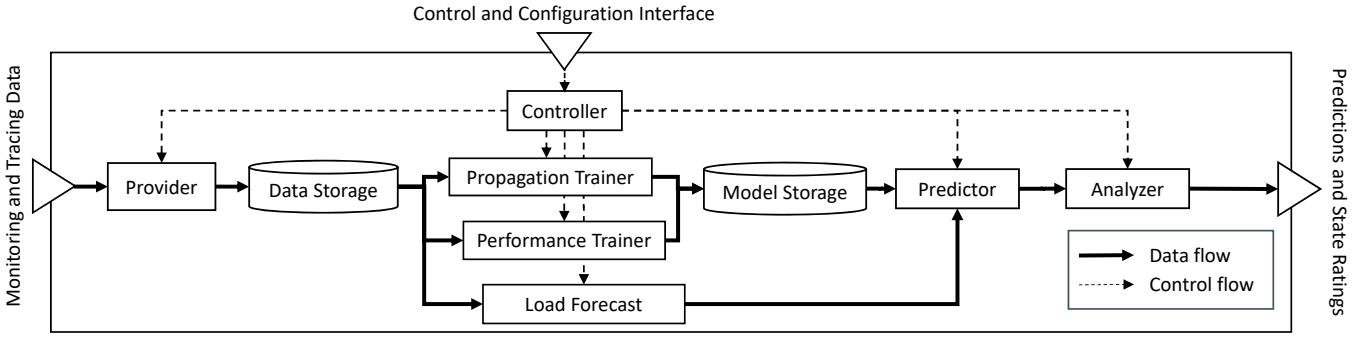

Figure 1: Architecture overview of the SuanMing framework.

metrics influencing the service performance including parameter values, deployment, and co-locations, or hardware specifications, can be added optionally. However, the target performance metrics — usually response times — are required as labels for the performance model training algorithms.

Both information is usually extractable from call traces, call stacks, or call trees. Therefore, the required monitoring and tracing data is obtainable in many state-of-the-art tracing tools, like Jaeger[1], Zipkin[2], Pinpoint[3], Dapper [27], or Kieker [29]. The current implementation supports the export formats of Zipkin, Pinpoint, and the proprietary format used in Huawei Cloud[4] (see Section 3.2).

2.3 Load Forecast

The *Load Forecast* component is responsible for forecasting the number of user requests in the next prediction interval. The forecast result is represented by a list U, where each entry $u_i \in \mathbb{R}_{\geq 0}^{m_i}$ represents the number of user requests to the m_i endpoints of service i. As our list S of observable services contains $s := |S|$ services, U contains s lists. In total, U contains m entries, where $m := \sum_{i=0}^{s-1} m_i$ is the total number of application endpoints.

However, the problem can be simplified by isolating each request type of U as a univariate time series and forecasting the expected requests independently. As there already exist many works capable of forecasting user behavior (e.g., [2, 17, 36]) we rely on GluonTS [1] in this work as they showed the best prediction performance in our prior analysis. However, due to the modular approach of SuanMing, this component can be seamlessly exchanged for another forecasting approach.

2.4 Propagation Trainer

The Propagation Trainer component is responsible for learning the propagation model describing how many additional internal inter-service requests are needed to process an incoming user request.

2.4.1 Formalization. We formalize this using the *propagation matrix* D. Each entry $d_{i,j} : \mathbb{R}_{\geq 0}^{m_i} \to \mathbb{R}_{\geq 0}^{m_j}$ in D represents a function mapping a number of incoming requests at service i to a number of requests to service j sent by service i. Both input and output of each propagation function $d_{i,j}$ are vectors containing numbers greater or equal to zero. The dimension of each vector is determined by the variable m_i, which represents the number of endpoints of service i.

Note that our model is able to assess the distribution of calls across the different endpoints of each service. This is useful for performance predictions, as different endpoints might have different performance metrics. For example, a web-server might take longer to show the dynamic login page, than to deliver a static index page. In total, the matrix D has s^2 entries overall, where s represents the number of services in the application.

The application topology modeled by D can also be visualized as a directed graph, where each node represents a service and an edge from node i to node j means service i sends requests to service j. In a graph representation, the edges between the nodes are the propagation functions stored in D. Consequently, we can describe paths in this graph by composite functions. For example, a path from i over j to k can be written as the composite function $d_{i,k} = d_{j,k} \circ d_{i,j}$ using the composition operator[5] $\circ$. A propagation model is *acyclic*, if no path $d_{i,i} = d_{i,j} \circ d_{j,k} \circ ... \circ d_{z,i}$ exists, where not at least one component function is a null function, i.e., a function that always returns zero. In other words, the topology graph omitting all edges of null functions must be acyclic. Analogously, a propagation model is cyclic, if at least one path $d_{i,i} = d_{i,j} \circ d_{j,k} \circ ... \circ d_{z,i}$ exists, where none of the component functions is the null function.

Our framework assumes that all valid propagation models are *regular models*. A propagation model is regular, if for every cycle $d_{i,i}$ and every input vector x the sequence

$$d_{i,i}(x)^n = \underbrace{d_{i,i} \circ d_{i,i} \circ ... \circ d_{i,i}}_{\text{n times}},$$

converges to zero for $n \in \mathbb{N}$ towards infinity. Such models represent topologies, where limited cycles, but no infinite loops exist. This means that, after a certain number of iterations, the application always terminates. All acyclic models are regular by definition.

2.4.2 Model learning. The task of the Propagation Trainer is to learn the propagation matrix D using historical data. To build this model iteratively, we initialize all functions $d_{i,j}$ as null functions, representing no dependency between the services i and j. After each observation interval, we update each function $d_{i,j}$, where a request between service i and j has been recorded. This step is relatively straight-forward to implement, as we simply need to

[5]Let f and g be functions, then the composite function $g \circ f$ is defined as $g \circ f = g(f(x))$

analyze the average behavior of the call tree of each request. The model learning still continuously updates the propagation model whenever new monitoring data becomes available in order to react to changes in application or user behavior.

2.5 Performance Trainer

As the final learning component, the performance trainer utilizes the information of the individual requests at each service endpoint to predict the performance of the individual endpoints and the resulting performance of the overall system. We assume that the performance is mainly dependent on the number and type of incoming requests to the service. This is a simplification made possible by the assumption of statelessness of modern microservice architectures [7, 31]. If all micro-services follow this requirement and are stateless, the performance of each request only depends on the availability of resources, which is dependent on the number and type of other requests arriving at the same processing resource, and the parameters of each request itself.

2.5.1 Formalization. The performance of a service i is described using the performance vector $p_i \in \mathbb{R}^{m_i \times r}$. To describe the performance of a service or endpoint, we use r different performance metrics of interest. These could be, for example, the average response time or the number of exceptions. These metrics are calculated for all m_i endpoints of service i. As a consequence, the service performance vector p_i of service i has $m_i \cdot r$ entries overall.

We obtain the requests arriving at i from the list of requests X calculated by the forecasting and the propagation model. Additionally, the framework allows adding additional features of arbitrary type α, e.g., the number of incoming requests on co-located or influencing services, the parameter distribution of the given requests, the measured resource utilization, a priority vector, etc. It is then dependent on the chosen modeling type, how the given auxiliary information is utilized. These auxiliary metrics α can be forecast using the standard load forecasting component also utilized for forecasting the request numbers (see Section 2.3) or other specialized forecasting or prediction engines. As the type and number of the auxiliary features is dependent on each service, α_i refers to the set of auxiliary features for service i. For adding additional important factors of the service performance, we consider the position of the service in the topology graph. We define z_i as a variable, which represents for a given service i the sum of all endpoints of all services, receiving calls from service i. A backend service only responds to incoming requests. Therefore, for a backend service b, $z_b = 0$. We assume, that its performance p_b only depends on the number of incoming requests x_b and the set of auxiliary metrics α_b. Therefore, the performance function f_b maps $m_b + |\alpha_b|$ input values to $r \cdot m_b$ performance metrics. For all other services i, we assume that the performance p_i additionally depends on the performance measures of all endpoints answering calls from i. Note that we do not make any assumptions about synchronous or asynchronous calls. It is up to the performance function to determine the actual impact of each influencing service.

Summarizing, for each service i there is a performance function $f_i : \mathbb{R}^{m_i}_{\geq 0} \times \mathbb{R}^{r \cdot z_i + |\alpha_i|} \rightarrow \mathbb{R}^{m_i \cdot r}$ which maps the requests list $x_i \in \mathbb{R}^{m_i}_{\geq 0}$, $r \cdot z_i$ additional influencing service performances, and $|\alpha_i|$ auxiliary input values to the predicted service performance $\tilde{p}_i$. We define F to be the list containing all s performance functions of an application.

Algorithm 1: Request propagation algorithm.

Input : Propagation matrix D, user requests U, threshold ϵ, number of services s, ordered set of endpoints $(m_1, \ldots, m_s)$
Output : List of total requests X.

```
X = U
X̄ = U
while not all numerical entries in X̄ equal 0 do
    X̂ = 0_s
    Set all entries x̂_i in X̂ to 0_{m_i}
    foreach x̄_i in X̄ do
        if x̄_i ≠ 0_{m_i} then
            foreach d_{i,j} in i-th row of D do
                x̂_j = x̂_j + d_{i,j}(x̄_i)
    X = X + X̂
    X̄ = X̂
    Set all numerical entries in X̄ lower than ε to 0
return X
```

2.5.2 Model learning. For learning performance models, historical training data is required. Hence, we have a training set L_i, consisting of t individual measurement intervals for each service i. The list of all L_i, form the set of the total training information available L. Each scenario shows us one input of $m_i + r \cdot z_i + |\alpha_i|$ performance-relevant features, together with r measurable performance metrics for each of the m_i endpoints of service i. Therefore, L_i is a matrix consisting of t measurement rows, with each row containing $(m_i + r \cdot z_i + |\alpha_i|) + (r \cdot m_i)$ values.

Due to the formalization and the modular design of SuanMing, we support multiple possible performance modeling and learning techniques. In this work, we will focus on black-box machine learning algorithms as they offer the most transferable and domain-independent performance. However, other approaches can be integrated as well, should the need arise for different or more expressive models. For example, we are simultaneously working on an approach using resource demand estimation [14, 28] for queueing theory, to model the performance of the system.

In this work, we approximate the performance functions in F using supervised machine learning to predict the r measurable performance metrics. As a consequence, we are able to train regression models, which approximate the performance function list F. As the performance functions f_i might be subject to change, we update F on a regular basis using measured performance data. Using this black-box modeling type, the additional performance indicators α, like deployment information or hardware specifications, can be easily added as no semantic meaning needs to be provided. Hence, all performance indicators that may seem relevant for the prediction of any of the r targeted performance metrics and that can be reliably monitored can be easily integrated into the performance predictor functions f_i.

Algorithm 2: Performance inference algorithm.

Input : List of services S, performance model F, list of requests X, list of additional metrics A.
Output : Predicted application performance $\tilde{P}$.

```
1  F, S, X, A = resolveCycles(F, S, X, A)
2  P̃, S' = ∅
3  while S' ⊊ S do
4      foreach i ∈ S \ S' do
5          Let d_i be the set of dependencies of service i
6          if d_i ⊆ S' then
7              P̃_i = {p̃_j ∈ P̃ | j ∈ d_i}
8              p̃_i = f_i(x_i ∪ α_i ∪ P̃_i)
9              P̃ = P̃ ∪ p̃_i
10             S' = S' ∪ i
11 return P̃
```

2.6 Predictor

The Predictor is one of the main components of the SuanMing framework. It combines the user requests forecast U with the propagation model D and the performance prediction model F in order to make a prediction about the future state.

2.6.1 Request propagation algorithm. The first step is to calculate the number of requests arriving at each service in the given time interval, i.e., propagate the user requests through the system. Given the service dependencies captured by D and the predicted user requests U, we want to predict how the requests are forwarded through the application. We propose an iterative algorithm that works with any regular model D.

Algorithm 1 takes the user request list U, which has been generated by the forecasting engine, as an input and returns the request list X, which contains the total number of predicted incoming requests in the next prediction interval for all services and endpoints. The list X can be seen as the sum of the incoming user requests U and their generated internal calls. Hence, at the beginning of the algorithm, the values of U are assigned to X. Then, the algorithm iteratively calculates the resulting internal calls starting in line 3. The list $\bar{X}$ represents the requests, which need to be forwarded in the current iteration of the algorithm. Once the loop terminates, there are no more requests to forward and we can return the total list of requests X.

In the inner loops of lines 6 and 8, line 9 evaluates the propagation functions $d_{i,j}$ are evaluated for each service i and all target services j, if i forwards requests. The newly generated internal requests are stored in the temporary support list $\hat{X}$, which gets filled with zeros at the beginning of each iteration (line 4 and 5). After iterating all services, the newly generated requests $\hat{X}$ are added to the total numbers of requests X and need to be considered for the next iteration. Hence, $\bar{X}$ gets set as $\hat{X}$. As already stated earlier, requests do not need to be integers, as a service can also call another service with a probability of, e.g., 80%. Therefore, functions $d_{i,j}$ are defined on non-negative real numbers for input and output.

To prevent an infinite loop, the threshold parameter ϵ is used. It represents a lower bound, which is applied to $\bar{X}$ and all numerical entries which are smaller than ϵ will be set to 0. This guarantees termination of Algorithm 1 for regular propagation models and enables it to still deal with cyclic topologies, given that they are regular. Additionally, Algorithm 1 is easy to parallelize, as the only synchronized calls are the commutative addition in line 10 and the assignment in line 11. This enables a very high scalability, even for increasing topology sizes as they can be processed independently.

However, if D is linear and acyclic, we can improve the scalability of the request propagation algorithm even more. A propagation model D is considered *linear*, if every propagation function $d_{i,j}(x)$ within D can be written in the form of $d_{i,j}(x) = c_{i,j} \cdot x$ with $c_{i,j}$ being a constant matrix. The composition of two linear functions $d_{i,k} = d_{j,k} \circ d_{i,j}$ is always also a linear function: $d_{i,k}(x) = c_{i,k} \cdot x$. This can be proven with the following equivalent transformation:

$$
\begin{aligned}
d_{i,k}(x) &= d_{j,k} \circ d_{i,j} && \mid \text{Definition of function composition} \\
&= c_{j,k} \cdot (c_{i,j} \cdot x) && \mid \text{Associative property} \\
&= (c_{j,k} \cdot c_{i,j}) \cdot x && \mid \text{Substitution: } c_{i,k} = c_{j,k} \cdot c_{i,j} \\
&= c_{i,k} \cdot x
\end{aligned}
$$

□

Additionally, if a service i receives calls from multiple origins, the total number of incoming requests is the sum of all inbound request flows. Hence, the vector of incoming requests x_i can be written as the sum of multiple linear propagation functions. We further know that the resulting request list X is the sum of the user requests U and the internal calls, while every internal call is originated by a user request. From these properties follows that every vector x_i is a linear superposition of the entries u_i of the user request list U. With that, we are able to calculate every x_i with a single matrix multiplication. Therefore, for linear and acyclic propagation models, the iterative calculations from Algorithm 1 can be summarized into s independent matrix multiplications. Depending on the size of an application topology, this might further improve the run-time and hence the scalability of the request propagation.

2.6.2 Performance inference algorithm. Finally, after we predicted the number of requests at each service i, the performance prediction algorithm infers $\tilde{p}_i$ for the given state.

Algorithm 2 iteratively calculates the performance $\tilde{p}_i$ for each service i by starting with all backend services and going backwards through the application topology. It requires the performance model $F = (f_1, \ldots, f_s)$, containing all performance inference functions learned in Section 2.5, the list of total requests for each service X= $(x_1, \ldots, x_s)$, and the list of additional metrics for each service A = $(\alpha_1, \ldots, \alpha_s)$. It is assumed that A is known using forecasting techniques and historical measurements of the available variables and can be treated analogously to X.

In line 1, Algorithm 2 first ensures that the application is acyclic, and then continues to initialize the performance vector P and the set of processed services S'. Then, it iterates through all services, until S' is no longer a proper sub-set of S, i.e., until every service has an associated performance prediction. For an unprocessed service i, we calculate the list of required performance values $\tilde{P}_i$ that are necessary for processing that service's performance prediction function f_i in line 7 based on the set of services d_i that i depends on. In line 6, it is determined whether $\tilde{P}_i$ can be calculated, i.e., whether

all influencing service metrics are already available. If so, the performance inference model is queried and stored in lines 8 and 9, and the service i is added to the list of processed services. Similarly to Algorithm 1, the calculation of the individual performance metrics is highly parallelizable for large topologies, improving the scalability of SuanMing.

Note that for any backend service b, $d_b = \emptyset$. Therefore, as $\emptyset \subseteq S'$ is always true, including $\emptyset \subseteq \emptyset$ in the first iteration, all backend services are added in the first iteration of the loop, as their performance only depends on the request list X and the additional metrics A. Hence, for acyclic graphs, Algorithm 2 is guaranteed to terminate. For regular and other cyclic models, Algorithm 2 does not terminate as all services of the cycle can not be calculated as long as their influencing services are not known.

Therefore, line 1 of Algorithm 2 refers to a heuristic capable of resolving cycles from the application model. One heuristic sets the performance values of all services of the cycle to ∞. This obviously results in false positives; however, all affected services will be post-processed in Algorithm 3, which is capable of solving cycles and therefore filters the false positives. Other possible heuristics include ignoring the dependency which generates the fewest calls for a given input list, or to contract all affected services into one hyper-service. While the former might reduce the prediction accuracy of an individual service, the latter lacks the granularity to pinpoint a specific service.

2.7 Analyzer

The final component focuses now on the analysis of the predictions produced in the previous steps. Therefore, the main aim of this component is to classify the severity of the predicted performance problems and to deliver explanations for the performance prediction in order to foster actionable insights. For example, two services a and b might experience a predicted performance degradations as their response times increase. However, as service a calls service b in order to answer its request, its performance is only degraded due to the increased waiting times at service a. Therefore, by addressing the performance degradation of b, we automatically also solve the performance problems of a.

In our case, the goal of the system operator is to define target ranges for every performance metric of user endpoints in advance. SuanMing is then tasked with supervising these ranges and alerting the operator if one value is predicted to exceed its target range in the near future. This translates into a binary classification problem. The classification is done using the predicted performance vector $\tilde{p}_i$ for every service i and comparing it to the defined target threshold t_i for each of the r performance metrics. The approach of binning performance metrics into different classes was already shown to be suitable for related performance problems [4].

However, as SuanMing focuses on explainable predictions, a simple problem classification does not suffice. Therefore, if a service is expected to perform worse than the defined threshold, the analysis component needs to pinpoint the service responsible for the anticipated performance problem in order to offer solutions on how to avoid it. In order to do so, Algorithm 3 calculates and returns the list of root-cause services R, responsible for the predicted performance problem. Based on the computation of R, an operator

Algorithm 3: Root-cause inference algorithm.

Input : Predicted performance $\tilde{P}$, performance thresholds T, performance model F, list of requests X, list of additional metrics A.
Output : Predicted application performance $\tilde{P}$, List of root-cause services R.

1 $R = \emptyset$
2 **foreach** $\tilde{p}_i \in \tilde{P}$ **do**
3 **if not** satisfies$_{t_i}(\tilde{p}_i)$ **then**
4 Let $\tilde{P}_i$ be the set of predicted dependent performance metrics of service i
5 $\tilde{P}'_i = \bigcup_{\tilde{p}_j \in \tilde{P}_i} \min \tilde{p}_j, t_j$
6 **if not** satisfies$_{t_i}(f_i(x_i \cup \alpha_i \cup \tilde{P}'_i))$ **then**
7 $R = R \cup i$
8 **if not** confident$(\tilde{P}, R)$ **then**
9 **return** $\emptyset, \emptyset$
10 **return** $\tilde{P}, R$

can specifically target all necessary services, e.g., by up-scaling all services in R, in order to avoid the predicted performance problem before it happens using minimal resource effort.

Similarly to Algorithm 2, Algorithm 3 requires the performance model F, the list of requests X, and the list of additional metrics A. In addition, Algorithm 3 requires the performance predictions $\tilde{P} = (\tilde{p}_1, \ldots, \tilde{p}_s)$, i.e., the output of Algorithm 2, and the defined performance thresholds $T = (t_1, \ldots, t_s)$ for each service.

Algorithm 3 utilizes the helper function satisfies$_{t_i}(\tilde{p}_i)$, to check, whether a performance prediction $\tilde{p}_i$ of service i violates any of its defined thresholds t_i. The function returns true, if any of the r components in the predicted performance vector $\tilde{p}_i$ is higher than its defined threshold t_i, i.e., satisfies performs an element-wise greater-than comparison.[6] If a service i is detected to violate any of its performance thresholds t_i in line 3, we calculate the all-fine performance vector $\tilde{P}'_i$ for service i in line 5. This is done by lowering all threshold-exceeding values to the defined threshold, and therefore simulating a normal behavior of all influencing services. If i still violates its threshold after all influencing services respond normally, it is added to the list of root-cause services R as i itself is responsible for the performance problem. On the other hand, if all performance predictions in $\tilde{p}_i$ fall below their respective threshold in t_i after all influencing services respond within their given boundaries, then it can be concluded that the performance problems of i can be fixed by fixing the performance problems of its influencing services. Hence, i is not considered to be a root-cause service.

After the calculation of R is done, the Analyzer finally conducts a confidence check of all its predictions in line 8. This step is necessary to avoid inaccurate performance predictions, especially due to lack of training data or model inaccuracies. Therefore, SuanMing enables the Analyzer component to scrap all performed calculations based on a configurable confidence function. If the confidence check fails,

[6] Without loss of generality, we assume that a threshold is always an upper bound for what is acceptable. If a threshold is set as a lower bound, one can simply negate the value and all of its predictions.

Algorithm 3 does not return any prediction. If the check succeeds, Algorithm 3 returns the application performance prediction $\tilde{P}$, as well as the list of responsible root-cause services R.

In this paper, our confidence value is based on the accuracy of the forecaster, as all model predictions are dependent on the forecasting accuracy and as it gives good insight into the general model accuracy. We use the coefficient of variation of the output distribution of the GluonTS [1] forecast as our measure of confidence. Hence, in the following evaluation, SuanMing starts to deliver root causes and ratings after the described coefficient of variation falls under 0.15. This value was chosen empirically after preliminary analysis in our test environment. However, in future work we plan to extend the confidence analysis to compare the respective prediction with actual measurements from the last intervals in order to rate the confidence of all component predictions.

As Algorithm 3 iterates over all performance predictions only once, it is guaranteed to terminate in linear time on both cyclic and acyclic topologies. Additionally, as all other algorithms presented in this paper, Algorithm 3 is highly parallelizable, improving the scalability of SuanMing.

2.8 Summary

To summarize, SuanMing relies on two fundamental models influencing the prediction power of the algorithm: the propagation model D and the performance inference model F. In the first phase, predicted user requests are forwarded through the application. In the second phase, the performance of each service, starting with the backend services, is determined by backpropagating the performance through the application. These two steps are able to deliver an accurate prediction of an arbitrary set of r performance metrics of interest – provided these performance metrics are captured by the monitoring infrastructure. Combined with an accurate forecast of the future user behavior, the models are able to predict the future state of the system.

Based on the predicted future state, an operator is then able to define target thresholds for every performance metric, either for all services or for sub-set as a priority list of supervised endpoints. If any of these priority endpoints is expected to miss its target, SuanMing is able to alert the operator and deliver a list of responsible services for an explainable performance prediction. Hence, fixing (e.g., by adding resources) the responsible services prevents the anticipated performance problems before they occur. If required, even without user facing performance degradation, SuanMing can still identify backend or intermediate services that experience performance degradation.

SuanMing is designed to work lightweight and fast on top of online cloud measurement infrastructures, does not need prior application knowledge, is scalable for large applications, and is able to deal with arbitrary kinds of acyclic and regular topologies. Furthermore, SuanMing is designed as a framework offering several extension points for the extension of all learning and prediction modules.

3 EVALUATION

For evaluating the effectiveness and the performance of SuanMing, we pose ourselves the following research questions (RQs):

- **RQ1**: *How do different regression modeling approaches compare for modeling the performance of an individual service?*
- **RQ2**: *Can SuanMing accurately predict the propagation of performance degradations between different services?*
- **RQ3**: *How do different forecasting horizons affect the performance of SuanMing?*
- **RQ4**: *Is the overhead of model training and performance inference feasible for online environments?*
- **RQ5**: *Is SuanMing effortlessly portable to different applications and monitoring environments?*

All results in the context of answering these research questions are published and can be replicated using a CodeOcean capsule[7].

3.1 TrainTicket

Our first experiment to answer the formulated research questions focuses on TrainTicket [35]. TrainTicket is a representative microservice application consisting of 42 different services concerned with the administration, searching, and booking of train tickets. Due to the number of services in the application and the depth of the call chains, it is suitable for demonstrating the performance propagation and pinpointing capabilities of SuanMing.

3.1.1 Experiment setup. We deploy each TrainTicket service in a single docker container deployed on an HPE ProLiant DL360 Gen9 cloud server equipped with an Intel® Xeon® E5-2640 v3 CPU, 32 GB of RAM, running Ubuntu 18.04 with Docker 18.09.7. All containers are resource-limited in order to minimize performance interference between the services. The SuanMing implementation is run on an HPE ProLiant DL20 Gen9 with an Intel® Xeon® E3-1230 v5 CPU, 16 GB of RAM, and running the same Ubuntu and Docker 18.09.7 environment. The TrainTicket services are monitored using the Pinpoint[3] monitoring framework deployed on a different Virtual Machine (VM) equipped with 2 CPU cores and 16 GB of RAM, running on a third host machine.

Lastly, we used an additional machine for emulating users visiting the TrainTicket website. We use the HTTPLoadGenerator [30] to generate a periodically increasing and decreasing amount of users on the system. Upon visiting the site, users log in, solve a captcha, search for a set of possible trains on a route, reserve and buy tickets, as well as collect and check-in the booked tickets. Our users randomly send incomplete or faulty data when searching or booking, in order to make the overall user behavior less predictable. In total, 26 services and 58 service endpoints are involved in processing the user requests. The base load varies between 3 and 22 requests per second. We set the prediction interval to five seconds, i.e., the experiment time is divided into intervals of five seconds. Therefore, our models generate every 5 seconds a new prediction about the next interval, resulting in a total of 676 analysis intervals with corresponding predictions and 700 total training intervals. As we have 58 endpoints in the application, SuanMing trains 58 machine learning models, using at least 1 and at most 12 features, depending on the position of the endpoint in the graph. In addition to this relatively low base load, we now specifically overload one of the backend services (`train`) with 300 requests per second in order to evaluate how the created performance degradation propagates

[7] https://doi.org/10.24433/CO.8530346.v3

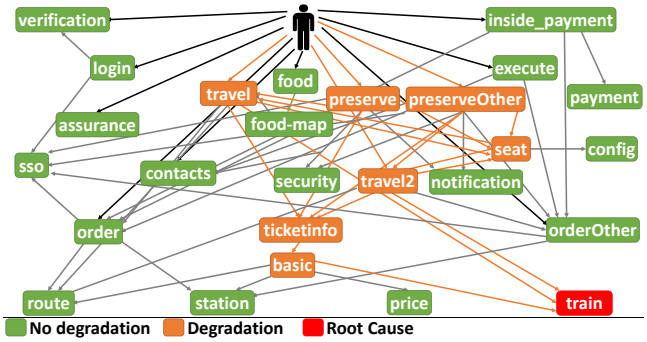

Figure 2: Schematic application state at measurement interval 680, i.e, 3400 seconds. Each box represents a service, an arrow depicts user or inter-service calls.

through the application and whether SuanMing is able to correctly predict and pinpoint the anticipated performance degradation. As we want to evaluate how this impacts the prediction performance on the `travel` service, we furthermore ensured that the load on `travel` was more or less constant, during the whole experiment (compare Figure 3).

Concerning performance metrics to measure and predict, we focus on one single metric in this evaluation, namely the service response time. For the front-end services `preserve`, `preserveOther`, and `travel`, we define thresholds of 1500, 1500, and 600 ms as performance thresholds. Hence, if the average response time of `preserve` rises above 1.5 seconds in any given time interval, `preserve` is considered to experience a performance degradation. The goal of SuanMing is now to predict this degradation at least one interval before it happens. The thresholds of all endpoints are set to at least two times their normal response time during low load. For fast services with response times smaller than 30 ms (except `train`), we set the threshold to 110 ms. As the propagation matrix of TrainTicket is linear, we can apply the linear propagation algorithm discussed in Section 2.6.

3.1.2 Performance propagation. Figure 2 shows a schematic overview of all 26 involved service instances and their connections. Figure 2 shows the system status during a performance degradation at the `train` service (red). We observe that the performance degradation propagates as expected through the application. Although the front-end services `travel`, `preserve`, and `preserveOther` only receive a very limited number of requests (8.8 requests per second), the average response time of these services increases massively after the backend service `train` decreases its performance. We observe performance degradations at several services (orange), all of which can be avoided, by addressing the problem at the `train` root-cause service (red). In the following, we analyze if SuanMing is able to correctly predict, detect, and propagate these performance degradations, as well as pinpoint the respective root-cause.

3.1.3 Regression analysis. For answering **RQ1**, we first analyze the performance of different regression models on the `/travel/query` endpoint. We focus on `/travel/query` as the `travel` service is the frontend endpoint and therefore relevant for user-experience. The goal of all modeling approaches is to predict the performance degradation at the `train` service and to propagate the performance problems up to the `travel` service. As `travel` itself is not experiencing a high base load, this assesses the model capabilities to correctly propagate the performance degradation to the system.

The black line of Figure 3 shows the average response time for a request at the `/travel/query` service over time, i.e., the time one user has to wait for the response of a search for possible trains. At the same time, the gray background curve depicts the number of requests arriving at `travel`. We observe, that the observed response time spikes are not correlated to the number of incoming requests at `travel`, but are due to the poor performance of other services.

In this work, we compare four different machine learning models: (1) Random Forest Regression (RF), (2) k-Nearest Neighbor Regression (KNN), (3) Bayesian Ridge Regression (Bayesian), and (4) Support Vector Regression (SVR), all provided by the Scikit-learn library [8]. All models are trained using 6-fold cross-validation using out-of-sample forecast evaluation [3] in order to optimize their hyper-parameter settings by performing a grid-search on a set of 3 to 5 hyper-parameters to minimize the classification error.

We observe that almost all modeling approaches depicted in Figure 3 closely resemble the anticipated load spikes. The random forest seems to perform best, as support vector regression tends to continuously underestimate the load spikes, Bayesian ridge regression tends to overestimate the low load phases, and k-nearest neighbor occasionally massively overshoots the predictions. However, generally, all modeling approaches are able to predict the performance degradation at `/travel/query`, although the incoming load intensity is relatively stable. Note that the measurement and the prediction curves were aligned for better visibility.In a live system, the prediction curves would rise *before* the measured response time increases, as it is necessary to enable the mitigation of the degradation.

Figure 3 additionally depicts the evolution of the regression model over time. The dashed line at 505 seconds resembles the confidence threshold to be passed for the first time. Hence, SuanMing did not output any degradation predictions before that time, as the model confidence was too low. This makes sense, as all SuanMing models start without any application information or a-priori knowledge and need some time to learn and adapt to the TrainTicket application. The chosen threshold itself is also reasonable, as the regression models are fairly inaccurate during the first experiment phase. As the confidence is calculated based on the forecast, it is model-agnostic and can be applied to all model types at the same time. For the presented experiment, the threshold was passed after 101 intervals or 505 seconds.

After the confidence threshold is passed, the Random Forest regression curve closely fits the measured performance, although the response time fluctuates heavily between 200 ms and over 3000 ms. However, we observe that the models continuously learn and adapt to the incoming measurements. For example, the load peak right after 2500 is lower than anticipated, therefore, Random Forest and k-nearest neighbor curve overshoot the expected response time during that degradation. Following, the models adapt to these changes and lowers its predictions for the succeeding intervals in order to better resemble the measurements. Generally, these small prediction errors lead to a relatively high regression error (compare Table 1), but have only minor effects on the classification accuracy.

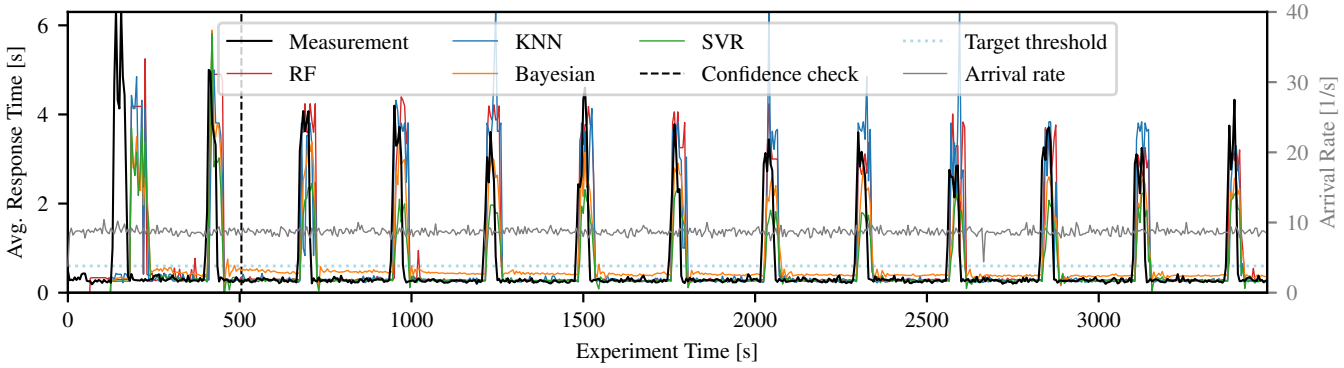

Figure 3: Measured and predicted response times at `/travel/query`, together with the incoming request rates.

Table 1: Model accuracy comparison for the TrainTicket service.

Approach	Example Endpoint `/travel/query`								Application-Wide		Overhead	
	sMAPE	MAE (ms)	TP	FP	TN	FN	Accuracy	F1 Score	Accuracy	F1 Score	Training	Prediction
RF	**0.271**	331	63	33	481	22	0.908	0.696	0.977	**0.626**	6.243s	0.226s
KNN	0.282	337	64	31	483	21	**0.913**	**0.711**	0.975	0.594	3.102s	0.135s
SVR	0.290	**288**	64	31	483	21	**0.913**	**0.711**	**0.978**	0.610	17.405s	0.092s
Bayesian	0.500	357	64	38	476	21	0.902	0.684	0.955	0.484	**2.915**s	**0.067**s
Mean Regr.	0.791	558	37	287	227	48	0.441	0.181	0.931	0.203	–	–
All-green	–	–	0	0	514	85	0.858	0.000	0.973	0.000	–	–
All-red	–	–	85	514	0	0	0.142	0.249	0.027	0.052	–	–
Random	–	–	46	265	249	39	0.492	0.232	0.497	0.050	–	–
OptimalRF	0.198	221	75	13	501	10	0.962	0.867	0.987	0.772	–	–

3.1.4 Classification analysis. In this section, we now evaluate the performance of SuanMing in more detail to answer **RQ1** and **RQ2** and to analyze the cost-performance trade-off of SuanMing. Table 1 shows different regression and classification scores of different model types summarized over the course of the whole experiment.

We report the mean absolute error (MAE) and the symmetric mean average percentage error (sMAPE) [10] of the regression predictions at each point. As the predicted response time of each service is then translated into a classification (healthy or not healthy) using the defined thresholds, we further study the classification performance of the individual approaches. We analyze true positive (TP), false positive (FP), true negative (TN), and false negative (FN) scores, as well as the resulting accuracy and F1 scores. Additionally, we report the *global* classification metrics, i.e., the accuracy, and the F1 score for all 58 service endpoints combined. The values reported in Table 1 all refer to the intervals after the model confidence threshold has passed, i.e., excluding the first 505 seconds.

For comparison, we added a set of different baseline approaches. First, we add *Mean Regressor* (Mean Regr.), a regression approach that always predicts the mean of all previously observed values. The *All-green*, *All-red*, and *Random* approaches are classifiers that always predict green, red, or randomly for each of the experiment intervals. Finally, we also add a variant of the Random Forest predictor that receives perfect load forecasts (OptimalRF). This approach assumes that the given forecast is perfect (i.e., it works a-posteriori and not online), and therefore helps at determining the impact of the forecasting error.

In total, the used experiment contains 85 intervals experiencing performance degradation at the `/travel/query` endpoint. Adding the other endpoints, a total of 933 performance degradations were recorded. In comparison to the total amount of intervals, this is a relative share of 14.2% for `/travel/query`, or 2.7% for the total application. This ratio is representative, as in practice performance degradations are expected rather infrequently.

Generally, we observe that all regression algorithms depicted in Table 1 are generally capable of capturing the performance behavior of `/travel/query`. The most notable difference is with the Bayesian Ridge Regression, which performs poorly on the regression metrics and has a higher false positive rate than the other approaches. While Random Forest performs best for sMAPE, Support Vector Regression shows a lower MAE and ties with k-Nearest Neighbor for accuracy and F1 score on the `/travel/query` endpoint. However, when we analyze the global score, Random Forest performs slightly better than Support Vector Regression regarding the F1 score. As the global F1 is our main metric of interest, and as the increased training of support vector regression speaks against it, we will restrict to Random Forest Regression for the rest of the

analysis. These results are in line with our analysis of the regression curves in Figure 3 and answers **RQ1** and **RQ2**.

When comparing with the baseline approaches, we observe that all modeling techniques consistently outperform the given baseline techniques. While the poor performance of All-green, All-red, and Random is expected, the Mean Regressor baseline also achieves significantly lower scores. Note that the high accuracy of the All-green and the Mean Regressor approaches is due to the large amount of true negative intervals in the experiment.

Finally, we observe that the performance of the Random Forest regressor can be significantly improved by eliminating the forecasting error with the optimal forecast. Although this approach is not realistic to apply in practice, we can conclude that Random Forest is able to correctly model the performance behavior of the system, given the correct number of incoming requests. This shows the benefit of the modular architecture of SuanMing, as the forecasting engine could easily be exchanged if more accurate forecasts were required.

3.1.5 Root-cause detection. As Random Forest is able to predict the performance degradation at the `travel` service, we now analyze the list of root-cause services returned by Algorithm 3. For the scenario shown in Figure 2, SuanMing returns a list of `train`, `travel`, `travel2`, `basic`, and `ticketinfo` as problematic or root-cause services which need to be addressed in order to solve the performance degradations in the system. However, by halving the inserted performance thresholds $\tilde{P}'_i$, the list is reduced to contain only `train`. This shows that SuanMing is able to pinpoint `train` as the respective root-cause service, but also identifies additional services that require attention. We hypothesize that these inaccuracies are due to the lack of training samples during high load phases, as performance degradations at `train` are always accompanied by performance degradations in the shown scenario.

3.1.6 Overhead analysis. In this section, we assess the overhead of model training and prediction in order to demonstrate the feasibility of using SuanMing in an online environment. To that end, Table 1 additionally depicts the maximum time required for training the regression models together and the mean prediction time per interval. We analyze the maximum training time, as the time required for training the regression models increases with the amount of monitoring data available. The prediction time includes both the execution of the request propagation and the performance inference for all endpoints of the service in the Predictor component and is averaged over all predictions.

We observe that all models are able to train all regression models in a matter of seconds and deliver predictions in less than a second. Both timescales are assumed to be fine in an online environment, and prediction times of less than one second are sufficiently fast for prediction intervals of 5 seconds. We note that Bayesian Ridge Regression is by far the fastest of all approaches. However, this is offset by its relatively poor prediction accuracy. Therefore, while the choice of the best-suited modeling type is up to the user, we still recommend using the Random Forest approach. This answers **RQ4**, as all assessed modeling types qualify for execution in online environments.

Table 2: Accuracy comparison on the TeaStore experiment.

Approach	`/webui/main`		Application-wide	
	Accuracy	F1 Score	Accuracy	F1 Score
SuanMing	0.826	0.816	0.913	0.693
Mean Regr.	0.520	0.681	0.802	0.574
All-green	0.486	0.000	0.857	0.000
All-red	0.514	0.679	0.143	0.251
Random	0.504	0.495	0.503	0.224

3.2 TeaStore

After we verified the capabilities of SuanMing on the TrainTicket application, we now move on to use SuanMing in order to predict performance degradations of a different application in a more realistic testing environment. As a second application, we use the TeaStore [31], a micro-service benchmarking application for buying tea, consisting of seven services. The users of TeaStore can log in, browse different categories and products, add and modify items in their shopping cart, and checkout by entering shipping and payment information. In contrast to the previous experiment, we now deploy TeaStore in a realistic, commercial cloud environment (Huawei Cloud[4]), and feed the available cloud monitoring and tracing into the SuanMing framework. Additionally, we increase the applied load pattern in order to represent realistic daily or weekly fluctuating workload intensities and to regularly overload the TeaStore application at specific services. Therefore, over the experiment duration of 6 hours, the workload intensity irregularly fluctuates between 1 and 140 user requests per seconds, while the target thresholds of the frontend service WebUI are set to 200 ms, and for all other services are set to 100 ms. This aims at verifying that SuanMing is not only able to deliver root-cause predictions under lab conditions, but also deliver accurate performance degradation predictions in a realistic cloud setting.

3.2.1 Classification performance. We evaluate SuanMing by analyzing the performance of the Random Forest modeling technique with different baseline approaches, similar to the previous experiment. Table 2 compares the classification metrics of SuanMing with other baseline approaches, focusing on the frontend service `/webui/main`. The shown values refer to after the confidence threshold was passed, which was after 13 intervals or 65 seconds.

Similar to our results on the TrainTicket application, SuanMing is able to outperform all baseline approaches in terms of accuracy and F1. However, we notice that the accuracy is lower, while the F1 score has increased in comparison to the TrainTicket environment. This is due to the increased number of performance degradations in the data set, and the correspondingly rising number of true positives. Nevertheless, SuanMing is able to achieve a global accuracy of over 91% with an F1 score of almost 0.7, which shows that the results of SuanMing are transferable to different applications and monitoring environments (**RQ5**).

3.2.2 Prediction horizon. One advantage of the SuanMing design is that the performance prediction models are time-agnostic and are sub-sequentially able to calculate performance predictions for any arbitrary application state. After analyzing the performance of

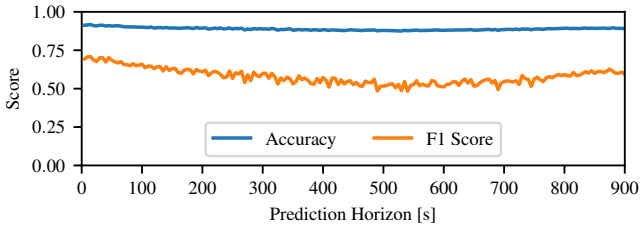

Figure 4: Global classification accuracy with increasing prediction horizons.

single-horizon predictions, i.e., the prediction for one interval or 5 seconds into the future, we now want to focus on increasing this prediction interval. The advantage of a higher prediction horizon is the fact that it allows for an earlier notification for an anticipated performance degradation and hence gives more time to address and mitigate the predicted problem.

Figure 4 shows the global classification accuracies averaged over the whole experiment duration for increasing prediction horizons. We observe that the F1 score is significantly impacted by an increasing horizon, while the reported accuracy stays almost constant. This is due to the fact that most services do not experience performance degradations, leading to a high amount of true negatives that can be accurately predicted by SuanMing. The F1 is mainly influenced by the predictions on the `webui` service as it was shown to be the bottleneck for the TeaStore application. Overall the curve drops almost linearly until a score of around 0.6 is reached after around 200s. Increasing the prediction horizon to over 500 seconds does not decrease the F1 anymore. While a performance prediction of 600 seconds in advance is theoretically possible, the F1 drops regularly below 0.6 for large horizons. It is then up to the user to decide if the accuracy drop is acceptable. However, if a small drop in prediction accuracy is acceptable, Figure 4 shows that SuanMing is able to predict performance degradations up to 100s to 200s seconds in advance. Generally, the prediction accuracy of large horizons is strongly dependent on the applied workload and whether the forecast engine is able to correctly predict the future load pattern. Unforeseen noise or load spikes makes this task increasingly difficult. We, therefore, conclude that SuanMing is able to arbitrarily increase the prediction horizon, but increasingly depends on accurate forecasts in order to perform its predictions (**RQ3**).

4 DISCUSSION

As our experiment in Section 3 shows, SuanMing and the accuracy of its performance predictions strongly rely on the applied forecasting technique. Due to the modular design of the SuanMing framework, it is possible to exchange or optimize forecasting engines, depending on the specific scenario or workload.

One of the main advantages of SuanMing is the high explainability of the results compared to state-of-the-art approaches. As we divide the prediction process into smaller steps, the root-cause prediction is augmented with interim results which can be consulted for further evaluations. In a practical use case, one can easily understand a predicted root-cause by consulting the load forecast, the request propagation model, and the resulting performance metrics.

In future work, we furthermore want to extend the current root-causing list by correlating predicted performance degradation with resource usage metrics in order to deliver root-cause not only on the service, but on the resource level. This can be done using the auxiliary metrics also used to improve the performance model accuracy. Furthermore, we plan to extend the dependency map of our approach, by taking deployment dependencies into account. This would also enable SuanMing to additionally detect performance degradations caused by an overloaded host, as successfully demonstrated by Lin et al. [22].

5 RELATED WORK

There are several works focusing on the explanation of performance degradation and namely root-cause analysis of cloud-hosted applications [19, 33]. Additionally, several works specifically targeting micro-service architectures have been proposed [20, 22, 32, 34, 35]. Notably, Microscope [22] uses causality graphs to pinpoint root causes for failures in microservice applications. In contrast to our work, they introduce the concept of non-communicating dependency in their dependency graph, e.g., via co-location. Other root case localization algorithms for micro-services based on call or dependency graphs were proposed by Wang et al. [32] and Wu et al. [34]. Zhou et al. [35] present an approach for fault analysis in microservice applications using tracing tools. However, none of these works focuses on predicting or forecasting future performance degradations.

On the other hand, works that focus on the prediction of performance problems require more data. For example, there exist works utilizing machine learning techniques, to detect or predict performance degradation of micro-service applications [12, 15]. However, they also utilize low-level monitoring data from the operating system or the hardware-level in order to calculate the performance predictions. While our approach is also able to utilize such additional monitoring data using the additional metrics α, SuanMing does not assume more than tracing data to be available. Similarly, other approaches based on rule-based detection [6, 16, 25] or architectural models [5, 23, 24, 26] rely on log data or other a-priori application knowledge, which is not available using the non-intrusive tracing.

Finally, a recent survey [13] analyzed approaches for the prediction of Service Level Objective (SLO) failures, of which the prediction of performance degradation as we define it in this work is a sub-set. They notice the lack of explainability in current works and therefore identified this as a research gap for future work [13], which is why SuanMing concentrates on delivering explainable and actionable predictions.

6 CONCLUSION

In this paper, we present SuanMing, a framework for modeling micro-service applications based on cloud monitoring and tracing data. The learned models do not require any a-priori application knowledge can be used to predict performance degradations as well as to pinpoint the responsible root-cause service in order to avoid the performance problem. We contribute a modular and mathematically formalized approach, that therefore enables easy adaptability and exchangability of individual components. The corresponding implementation is designed as a micro-service application as well.

Our evaluation shows that SuanMing is able to predict and explain performance degradations with an accuracy of over 90% on both the TrainTicket and the TeaStore micro-service applications. We furthermore assess that SuanMing is capable of delivering those predictions with a reasonable overhead in a realistic cloud environment. Finally, we observe that it is possible to receive performance degradation predictions several minutes in advance by increasing the prediction horizon, if accompanying accuracy decreases are acceptable. Therefore, SuanMing presents a first step towards autonomically supervising micro-service applications, in order to avoid performance degradations before they actually occur in the system.

REFERENCES

[1] Alexander Alexandrov, Konstantinos Benidis, Michael Bohlke-Schneider, Valentin Flunkert, Jan Gasthaus, Tim Januschowski, Danielle C. Maddix, Syama Rangapuram, David Salinas, Jasper Schulz, Lorenzo Stella, Ali Caner Türkmen, and Yuyang Wang. 2020. GluonTS: Probabilistic and Neural Time Series Modeling in Python. *Journal of Machine Learning Research* 21, 116 (2020), 1–6.

[2] Andre Bauer, Marwin Zufle, Nikolas Herbst, Albin Zehe, Andreas Hotho, and Samuel Kounev. 2020. Time Series Forecasting for Self-Aware Systems. *Proc. IEEE* 108, 7 (2020), 1068–1093.

[3] Christoph Bergmeir, Mauro Costantini, and José M. Benítez. 2014. On the usefulness of cross-validation for directional forecast evaluation. *Computational Statistics & Data Analysis* 76 (2014), 132–143.

[4] Ricardo Bianchini, Marcus Fontoura, Eli Cortez, Anand Bonde, Alexandre Muzio, Ana-Maria Constantin, Thomas Moscibroda, Gabriel Magalhaes, Girish Bablani, and Mark Russinovich. 2020. Toward ML-Centric Cloud Platforms. *Commun. ACM* 63, 2 (2020), 50–59. https://doi.org/10.1145/3364684

[5] Pedro Capelastegui, Alvaro Navas, Francisco Huertas, Rodrigo Garcia-Carmona, and Juan Carlos Dueñas. 2013. An online failure prediction system for private IaaS platforms. In *Proceedings of the 2nd International Workshop on Dependability Issues in Cloud Computing (DISCCO '13)*. Association for Computing Machinery, New York, NY, USA, 1–3.

[6] Alexander Clemm and Malte Hartwig. 2010. NETradamus: A forecasting system for system event messages. In *IEEE/IFIP Network Operations and Management Symposium (NOMS)* (2010), Yoshiaki Kiriha, Lisandro Zambenedetti Granville, Deep Medhi, Toshio Tonouchi, and Myung-Sup Kim (Eds.). IEEE, USA, 623–630. https://doi.org/10.1109/NOMS.2010.5488430

[7] Simon Eismann, Cor-Paul Bezemer, Weiyi Shang, Dusan Okanovic, and Andre van Hoorn. 2020. Microservices: A Performance Tester's Dream or Nightmare?. In *Proceedings of the 2020 ACM/SPEC International Conference on Performance Engineering (ICPE) (ICPE'20)*. ACM, New York, NY, USA, 12 pages. Acceptance Rate: 23.4% (15/64).

[8] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12, 85 (2011), 2825–2830.

[9] Maria Fazio, Antonio Celesti, Rajiv Ranjan, Chang Liu, Lydia Chen, and Massimo Villari. 2016. Open Issues in Scheduling Microservices in the Cloud. *IEEE Cloud Computing* 3, 5 (2016), 81–88.

[10] Benito E. Flores. 1986. A pragmatic view of accuracy measurement in forecasting. *Omega* 14, 2 (1986), 93–98.

[11] Martin Fowler. 2015. Microservice Trade-Offs. https://martinfowler.com/articles/microservice-trade-offs.html

[12] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. Seer. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 19–33.

[13] Johannes Grohmann, Nikolas Herbst, Avi Chalbani, Yair Arian, Noam Peretz, and Samuel Kounev. 2020. A Taxonomy of Techniques for SLO Failure Prediction in Software Systems. *Computers* 9, 1 (2020), 10.

[14] Johannes Grohmann, Nikolas Herbst, Simon Spinner, and Samuel Kounev. 2017. Self-Tuning Resource Demand Estimation. In *Proceedings of the 14th IEEE International Conference on Autonomic Computing (ICAC 2017)*. IEEE, USA, 21–26.

[15] Johannes Grohmann, Patrick K. Nicholson, Jesus Omana Iglesias, Samuel Kounev, and Diego Lugones. 2019. Monitorless: Predicting Performance Degradation in Cloud Applications with Machine Learning. In *Proceedings of the 20th International Middleware Conference* (Davis, CA, USA) *(Middleware '19)*. Association for Computing Machinery, New York, NY, USA, 149–162.

[16] Xiaohui Gu, Spiros Papadimitriou, Philip S. Yu, and Shu-Ping Chang. 2008. Online Failure Forecast for Fault-Tolerant Data Stream Processing. In *2008 IEEE 24th International Conference on Data Engineering*. IEEE, USA, 1388–1390.

[17] Nikolas Herbst, Ayman Amin, Artur Andrzejak, Lars Grunske, Samuel Kounev, Ole J. Mengshoel, and Priya Sundararajan. 2017. Online Workload Forecasting. In *Self-Aware Computing Systems*, Samuel Kounev, Jeffrey O. Kephart, Xiaoyun Zhu, and Aleksandar Milenkoski (Eds.). Springer Verlag, Berlin Heidelberg, Germany, 529–553.

[18] Pooyan Jamshidi, Claus Pahl, Nabor C. Mendonca, James Lewis, and Stefan Tilkov. 2018. Microservices: The Journey So Far and Challenges Ahead. *IEEE Software* 35, 3 (2018), 24–35.

[19] Hiranya Jayathilaka, Chandra Krintz, and Rich Wolski. 2017. Performance Monitoring and Root Cause Analysis for Cloud-hosted Web Applications. In *Proceedings of the 26th International Conference on World Wide Web (WWW '17)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 469–478.

[20] Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. 2019. Performance Modeling for Cloud Microservice Applications. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering (ICPE '19)*. Association for Computing Machinery, New York, NY, USA, 25–32.

[21] James Lewis and Martin Fowler. 2014. Microservices: a definition of this new architectural term. https://martinfowler.com/articles/microservices.html

[22] Jinjin Lin, Pengfei Chen, and Zibin Zheng. 2018. Microscope: Pinpoint Performance Issues with Causal Graphs in Micro-service Environments. In *Service-Oriented Computing*, Claus Pahl, Maja Vukovic, Jianwei Yin, and Qi Yu (Eds.), Vol. 11236. Springer International Publishing, Cham, 3–20.

[23] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Rui Xin. 2020. Predicting failures in multi-tier distributed systems. *Journal of Systems and Software* 161 (2020), 110464.

[24] Burcu Ozcelik and Cemal Yilmaz. 2016. Seer: A Lightweight Online Failure Prediction Approach. *IEEE Transactions on Software Engineering* 42, 1 (2016), 26–46.

[25] Teerat Pitakrat, Jonas Grunert, Oliver Kabierschke, Fabian Keller, and Andre van Hoorn. 2014. A Framework for System Event Classification and Prediction by Means of Machine Learning. In *Proceedings of the 8th International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS '14)*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), Brussels, BEL, 173–180.

[26] Teerat Pitakrat, Dušan Okanović, André van Hoorn, and Lars Grunske. 2018. Hora: Architecture-aware online failure prediction. *Journal of Systems and Software* 137 (2018), 669–685.

[27] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Technical Report. Google, Inc.

[28] Simon Spinner, Giuliano Casale, Fabian Brosig, and Samuel Kounev. 2015. Evaluating approaches to resource demand estimation. *Performance Evaluation* 92 (2015), 51–71.

[29] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. 2012. Kieker. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*. ACM, New York, NY, USA, 247.

[30] Joakim von Kistowski, Maximilian Deffner, and Samuel Kounev. 2018. Run-Time Prediction of Power Consumption for Component Deployments. In *2018 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, USA, 151–156.

[31] Joakim von Kistowski, Simon Eismann, Norbert Schmitt, Andre Bauer, Johannes Grohmann, and Samuel Kounev. 2018. TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, USA, 223–236.

[32] Ping Wang, Jingmin Xu, Meng Ma, Weilan Lin, Disheng Pan, Yuan Wang, and Pengfei Chen. 2018. CloudRanger: Root Cause Identification for Cloud Native Systems. In *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid '18)*. IEEE Press, USA, 492–502.

[33] Jianping Weng, Jessie Hui Wang, Jiahai Yang, and Yang Yang. 2018. Root Cause Analysis of Anomalies of Multitier Services in Public Clouds. *IEEE/ACM Trans. Netw.* 26, 4 (2018), 1646–1659.

[34] Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. 2020. MicroRCA: Root Cause Localization of Performance Issues in Microservices. In *IEEE/IFIP Network Operations and Management Symposium (NOMS)*. IEEE, Budapest, Hungary, 1–9.

[35] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2018. Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. *IEEE Transactions on Software Engineering* 1, 01 (2018), 1–1.

[36] Marwin Züfle, André Bauer, Nikolas Herbst, Valentin Curtef, and Samuel Kounev. 2017. Telescope: A Hybrid Forecast Method for Univariate Time Series. In *Proceedings of the International work-conference on Time Series (ITISE 2017)*. Springer, Berlin Heidelberg, Germany.