Performance Evaluation of a Legacy Real-Time System: An Improved RAST Approach

Juri Tomak(✉), Adrian Liermann, and Sergei Gorlatch

University of Muenster, Münster, Germany
{jtomak,a_lier03,gorlatch}@uni-muenster.de

Abstract. A challenging aspect in optimizing legacy distributed systems with strict real-time requirements is how to evaluate the performance of the system running in a production environment without disrupting its regular operation. The challenge is even greater when the System Under Evaluation (SUE) runs within a resource-sharing environment and, thus, is affected by the resource usage of other software running in the same environment. Current performance evaluation methods dealing with this challenge rely on data collected by Application Performance Monitoring (APM) tools that are not always available in existing systems and hard to establish when the system is already in production. In this paper, we improve the initial, proof-of-concept implementation of our RAST (Regression Analysis, Simulation, and load Testing) approach to evaluate the response time of a distributed system using the available system's request logs. In particular, we greatly improve the prediction model based on machine learning. Our use case is a commercial alarm system in productive use, developed and maintained by the GSelectronic company in Germany. We experimentally demonstrate that our improvements significantly enhance RAST's capability to adequately predict the system performance and verify the strict requirements on the response time. We make our model and software freely available in order to enable reproducing our experiments.

Keywords: performance evaluation · real-time requirements · regression analysis · simulation · distributed system

1 Introduction

Evaluating performance, e.g., the response time[1], of a legacy real-time software system that runs in a production environment is a great challenge as the system's regular operation is not allowed to be disrupted by the evaluation. A further challenge to performance evaluation is when the System Under Evaluation (SUE)

[1] Response time is the time interval between a sent request and the received response to it; it usually includes network latency and the request's processing time.

This work was supported by the DFG project PPP-DL at the University of Muenster.

J.-L. Guisado-Lizar et al. (Eds.): SIMUtools 2023, LNICST 519, pp. 18–33, 2024.
https://doi.org/10.1007/978-3-031-57523-5_2

runs within a resource-sharing environment, where the performance of the SUE is significantly affected by the resource usage of other software running in the same environment.

In our work, we are interested in evaluating the performance of existing, real-time software to decide what parts of it and how should be improved to comply with strict real-time requirements.

Current performance evaluation methods traditionally rely on data collected by Application Performance Monitoring (APM) tools [1,6,8,15]. These data usually contain low-level metrics of resource utilization, like CPU-, or RAM-utilization, or detailed information regarding network requests, like their size in bytes. The collected data are then used for creating a prediction model for the system's response time. However, we consider systems that often do not provide an APM and its integration is too complicated. Our target software only implements request logging, i.e., incoming network requests and their responses are logged, including their type, timestamp and executing thread. Beyond that, there is no further information regarding the requests, like the payload size.

As an alternative to using an APM, we introduce in [22] the RAST (Regression Analysis, Simulation, and load Testing) approach and its initial implementation as a proof of concept. The idea of RAST is to utilize log files (in the literature called request logs or access logs) that are typically created by the System Under Evaluation (SUE). In contrast to an APM, these log files provide only information about when a request was received by the system, when its processing was finished (or alternatively when the response was sent), and the request type. Using these data, RAST creates a prediction model for the time needed for processing network requests, e.g., HTTP requests, that the system processes. RAST automatically determines the optimal prediction model for the analyzed system using the provided request logs; it chooses the best regression algorithm via cross-validation of common regression algorithms, such as: Linear, Lasso, Ridge, Decision tree, and Elastic net regression. The prediction model is deployed in the Simulator of RAST, which simulates the system and takes the same requests as the SUE, and then uses the prediction model to predict the request's processing time and to delay the server's response accordingly. The Load Tester component of RAST is based on our load-testing approach [21]: it generates network requests, sends them to the SUE or the Simulator and measures the response time.

Our aim in presenting RAST is to provide a comprehensive and configurable toolset, encompassing common practices in log transformation, prediction model generation, Simulation, and Load Testing, to researchers engaged in performance evaluation of mission-critical legacy systems. Our initial version of RAST, introduced in [22], served as a proof-of-concept. This paper addresses its initial limitations with the following contributions:

- We implement common training data preprocessing methods, as well as automatic hyperparameter optimization via the grid search method [3]. To enhance flexibility, each individual method can be activated or deactivated via configuration settings.

- To assess the impact of each implemented method on the R^2 score [19], we conduct experiments using a commercial alarm system as a case study. Additionally, we identify the RAST configuration that yields the most substantial improvements over the previous version, achieving the highest R^2 score of 0.792 (with an ideal score of 1.0). Notably, this represents a remarkable 47% improvement over the best score of 0.63 obtained in the initial proof-of-concept implementation.
- To facilitate accessibility and reproducibility, we publish our improved, full RAST implementation on GitHub [20]. Additionally, we provide pre-trained models from our case study in both PMML and JSON formats, widely used data interchange formats for models. This further supports the reproducibility and reliability of our experiments.

In the rest of the paper, Sect. 2 provides an overview of our RAST approach and highlights the key features and improvements implemented in comparison to the initial, proof-of-concept version. Section 3 describes an alarm system's typical architecture and performance requirements and how our target alarm system of the GS company group works. Section 4 showcases various configurations of RAST and their influence on the achieved model's R^2 score. Section 5 presents our experiments with the new RAST implementation for evaluating the alarm system performance and finding the system's saturation point, i.e., the maximum number of Alarm Devices (ADs) that can be simultaneously handled while complying with the real-time requirements. Section 6 concludes the paper and outlines future work.

2 The RAST Approach: Overview and the Improved Implementation

Figure 1 illustrates the six components of RAST – our approach that combines Regression Analysis, Simulation, and load Testing to evaluate performance (in particular, the response time) of a production real-time system.

In this paper, we focus on our improvements to Pipeline A (2). For further details about the components (1), (3)–(6), refer to the caption of Fig. 1 and [22].

Figure 2 depicts two components – Log-Transformer and Prediction-Model-Creator – inside of Pipeline A of RAST and the data flow between the components, indicating our newly implemented components in bold font. The new components can be selectively activated or deactivated via the configuration file so that we can precisely study their influence on the prediction quality.

The component Log-Transformer (LT) in Fig. 2 creates a database with training data from the contents of the log files. The dedicated LT component allows the Prediction-Model-Creator component to become more generic, because processing the specific SUE details is offloaded to LT.

Training data are produced by the LT component and then are taken by Predictive-Model-Creator (PMC) that creates a prediction model by using regression analysis (Fig. 2). The goal of the prediction model is to predict the request's processing time by the system. This time depends upon the amount of concurrent requests and their types.

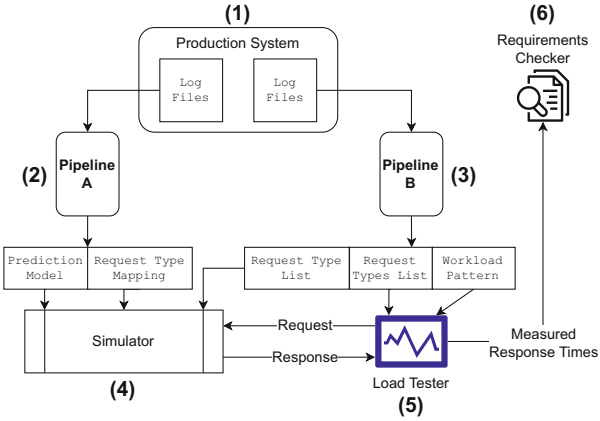


Fig. 1. The RAST components and connections: The analyzed system transfers its log files to Pipelines A and B that process the logs and produce the prediction model, the request type mapping, the list of all request types, and the pattern of system's workload; these are transmitted to Simulator and Load Tester. Load Tester produces a synthetic workload, sends requests to Simulator, measures the response times, and sends them to Requirements Checker that verifies if they comply with the performance requirements. When receiving a request, Simulator verifies request's type and transforms it to a numerical value for the prediction model. Simulator utilizes the prediction model for delaying the response for the time needed by the analyzed system for processing the corresponding request.

Our prediction model includes a regression algorithm with its parameters that are calculated by regression analysis (for example, linear regression and its coefficients). Our regression analysis employs a variety of machine-learning methods for predicting a continuous outcome variable using the values of one or several predictor variables [18].

It is essential for the choice of suitable predictor variables in our RAST approach that we can observe the variables at simulation run time in order to generate an input vector for our prediction model. While the target of our approach is the processing time, the predictor variables include the amount of concurrent requests at the beginning of the request, the amount of finished concurrent requests during the request processing, and the request's type.

Proper data preprocessing is an important factor in creating high-quality prediction models [10]. The PMC component implements two common data preprocessing methods: outlier detection and removal, and zero removal. The PMC component employs the standard deviation method for outlier detection and removal [4]: it calculates the mean and standard deviation of the processing times and it removes all values that lie from the mean more than three-times the standard deviation away. The PMC component offers two approaches for outlier detection: global and request. Global outlier detection considers all processing times whereas the request outlier detection, introduced in the new version, takes

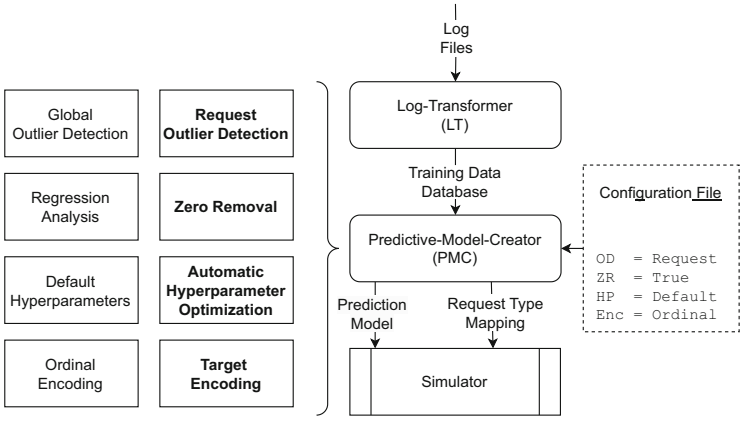


Fig. 2. Components of Pipeline A in RAST. Compared to the initial version [22], our new PMC component adds a request outlier detection method that is more precise than global outlier detection, adds a zero removal method, and performs automatic hyperparameter optimization. A configuration file enables choosing the preferred methods.

the processing times of each individual request type. In addition, the updated PMC component can perform zero removal which removes all database records with a processing time of zero, as they are often considered to be flawed measurements.

After removing outliers from the data, the PMC component compares common regression algorithms, called *estimators* in the following. Each estimator takes as input the training data and determines the optimal model parameters for a particular regression algorithm. This process in which the model parameters for the prediction model are learned via training is controlled by the values of hyperparameters. Hyperparameters are specific to the chosen estimator. Examples of such hyperparameters are the depth of a decision tree or the alpha value of ridge regression. The choice of proper hyperparameters has a great impact on the quality of a prediction model [3]. By default, RAST uses the default hyperparameters for the regression algorithms provided by the *scikit-learn* library [16], whereas in the new version, automatic hyperparameter optimization via grid search [3] can be activated. Grid search exhaustively searches for the optimal hyperparameters from a predefined set of hyperparameter combinations for a given estimator. We use Grid search in conjunction with five-fold cross-validation [19]. This involves performing cross-validation for each combination of hyperparameters and estimators, resulting in a score for each of them. We select the best estimator using the scores, and we export the best found estimator as a file, thus enabling the Simulator to use our prediction model.

Since many regression algorithms operate only on numerical data, PMC component modifies all textual requests in the training data to numeric values when loading the data. This modification assigns a unique number to each type of

request. Every assignment is memorized in a hash map, so generating the request type mapping. PMC can use two different ways to generate the unique number: by ordinal encoding or target encoding [17], the latter being introduced in the new version. Ordinal encoding is a method that assigns a numerical value to each distinct request type incrementally. The first request type is assigned the number one, the second is assigned two, and so on. However, when ordinal encoding is used with nominal data, such as request types, it introduces an order that does not exist in reality. For instance, a request type x_i is not inherently less or greater than a request type x_{i-1} or x_{i+1}, $\forall i \in (1, 2, ..., \text{number\_of\_requests})$. This limitation makes ordinal encoding less suitable for nominal data. For nominal data, other encoding techniques, such as target encoding, are more appropriate [17]. Target encoding calculates the mean value of all response times for each request type and assigns this value to the respective request type. To improve accuracy, an additional smoothing step is often applied by multiplying the mean value with the relative frequency of the request type. In contrast to ordinal encoding, target encoding takes the influence of the request type on the response time into account, resulting in a more accurate encoding. This consideration is expected to positively influence the prediction model's quality. After the completion of Pipeline A, our Simulator gets the prediction model and the request type mapping.

3 An Alarm System as the Use Case

Our use case is illustrated in Fig. 3: a typical production-quality alarm system. A so-called Alarm Device (AD) is installed at the customer's home. When detecting a breach of some safety criterion, e.g., burglary or fire, or an AD's own malfunction, AD sends an alarm message to the Alarm Receiving Software (ARS) that runs in the Alarm Receiving Center (ARC), a computer center that is the endpoint for the messages from ADs.

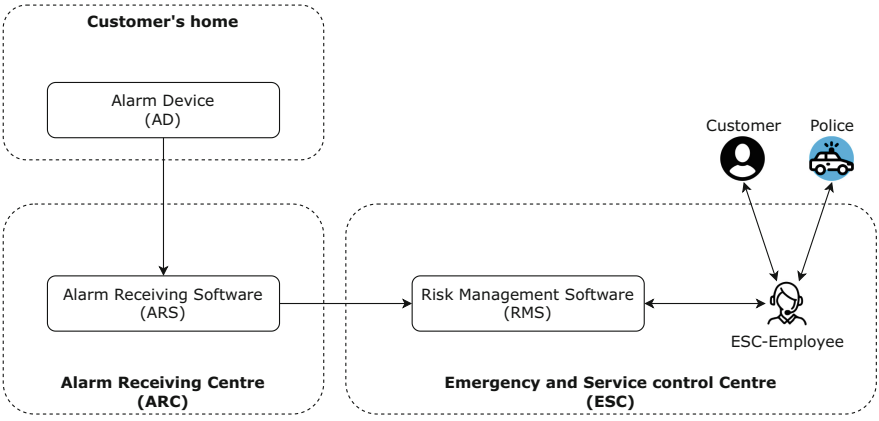


Fig. 3. A production-quality alarm system as our use case

ARS receives and processes alarm messages and then they are forwarded to the Risk Management Software (RMS) that runs in the Emergency and Service control Center (ESC). After an alarm message was processed by ARS, an acknowledgment is sent to the corresponding AD, such that no message resending is needed. The employees of ESC may take respective counter-measures as agreed with the customer.

Our use case in this paper is a production-quality software for alarm systems developed and maintained by the GSelectronic company [9] – one of the leading providers in Germany. It handles per day about 13.000 alarms received from more than 82.000 alarm devices working 24/7 all over Germany.

Nowadays, the software system supported by GSelectronic is a very large, quite complex, and not always well documented legacy distributed system that consists of about 80 executables and libraries written in C++, C#, BASIC, and Java, counting altogether more than $1.5 \cdot 10^6$ lines of code. The majority of the previous software developers are no longer available; furthermore, the system misses automated testing mechanisms, e.g., unit tests. In other words, we deal with the worst case of a legacy system, which is defined in the literature as "a large software system that we don't know how to cope with but that is vital to our organization" [14].

Therefore, our practical goal of performance evaluation for a productive alarm system poses several serious challenges:

1. Resource-sharing environment: the ARS is only one of altogether 12 server programs constituting the target SUE. Moreover, the system simultaneously runs anti-virus and backup software. These programs run in virtual machines (VM) and share system resources, e.g., CPU time, network bandwidth, memory, etc.;
2. Database-centric architecture: all servers' programs in the system use the same database instance, which can thus become a bottleneck;
3. The system has software bugs, is under continuous development, while the test environment is a legacy IT infrastructure.

Our measurements and company's statistics show that only 3% of the requests processed by the system are indeed processed by ARS, while the vast majority of requests go to other server software. Due to this high workload in background, one usually has to face unreliable performance measurements for the ARS. In the following, we show how our RAST approach can rectify this problem.

The timing behavior of an alarm system must follow the EN 50136 standard [7]. The *response time* between AD and ARS is the most relevant performance metric: it shows how timely the system responds to a request. Traditionally, according to the ISO/IEC 2382 standard [11], response time is defined as the time interval between a request and *any* response. However, the EN 50136 standard for alarm systems defines the response time as the time interval between an alarm message (request) sent and a *positive* acknowledgment (response) received, i.e., negative responses and timeouts do not count: ARS must meet the agreed

real-time requirements in presence of failures and high workloads. The precise requirement on the response time in the EN 50136 standard for alarm systems is currently: the arithmetic average of all response times taken in any time interval is not allowed to exceed 10 sec, and the maximum response time must be under 30 sec.

In this paper, our objective is to determine the *saturation point* of the ARS, i.e., the maximum number of simultaneous Alarm Devices (ADs) that can be handled by the system. In the practical sense, the GSelectronic company plans to upgrade about 25.000 outdated ADs to a modern version, so the company desires to have a confirmation in advance that the system will be able to handle the increased workload of modern devices. A modern AD sends requests for health checking much more frequently (every 20 to 90 sec compared to once per day); moreover, much more data are exchanged during each request.

4 Training the Optimal Prediction Model

In this section, we: a) describe the process of training the optimal prediction model, b) showcase the influence on the R^2 score with different configurations for outlier detection, encoding, and hyperparameters, and c) compare the results of the new RAST implementation described in this paper vs. the initial, proof-of-concept implementation presented in [22]. To ensure a fair comparison, we utilize the same estimators as in our initial implementation.

For our training, we utilize one node of the PALMA II ("Parallel Linux System for Muenster Users" abbreviated in German) supercomputer provided by the University of Muenster. The node is equipped with Intel Xeon Gold 6140 18C processors with 2.30 GHz of which we utilize 36 CPU cores as well as 192 GB of RAM. Our PMC component utilizes multiple cores at two different stages:

- When automatic hyperparameter optimization is active, the grid search is performed with 36 so called jobs, i.e., up to 36 hyperparameter/estimator combinations are evaluated in parallel;
- The regression analysis and cross-validation process is implicitly parallelized by the *scikit-learn* library.

To make our results comparable, we use the same request logs for both implementations which we compare. The logs include 5 out of 12 server programs of the system, as only these programs produce request logs. Each request processed by these programs generates two lines of text in the log file, one for the begin of processing a request, one for its end. One line of text in the log file is usually named a log entry. From the selected server programs, we use 180 log files from 30 days spanning a 13-month period between Dec 2020 and Jan 2022. The selected days capture all of the different business workflows performed by the company throughout the year, including monthly and yearly accounting tasks, daily operations such as alarm message processing and customer handling, and other relevant workflows. Altogether, our considered log files feature about $40{\cdot}10^6$ log entries. The amount of resulting rows in the training database is about half

of the amount of log entries, since one pair of log entries is used to calculate the processing time of a request, which means that our training database consists of around 20 million rows and 4 columns of data: the amount of parallel requests at the request beginning (PR 1 N), the amount of parallel requests finished during the request's processing (PR 2 N), the type of the request (CMD), and its processing time. We refer to the ordinal encoded request type as CMD O and to the target encoded request type as CMD T.

As an initial step of comparison between various configurations, we configure the PMC component with the exactly same settings as in our initial study (global outlier detection, no zero removal, and default hyperparameters). We do so to validate that our new RAST implementation produces identical results to its predecessor. Moreover, this provides us with a benchmark against which to measure our improvements.

Table 1 shows the estimators and their respective R^2 scores. We observe identical scores to our initial case study [22] with the DecisionTree Regression being the best and Ridge Regression having a lower score, by 0.089. Lasso and ElasticNet Regression are effectively useless with this configuration. Thus, our new implementation, when used with the predecessor's configuration, successfully produces identical results, demonstrating its successful implementation.

Table 1. Baseline R^2 scores of the estimators when using CMD 0, PR 1 N, PR 2 N as predictor variables, global outlier detection, and default hyperparameters. These scores are identical to the results of our initial study [22].

Estimator (regression type)	R^2 Score
Ridge	0.537
Lasso	−0.000
ElasticNet	−0.000
DecisionTree	0.626

Table 2 shows the improved R^2 scores for our next configuration of the PMC component: using request outlier detection, zero removal, and keeping the default hyperparameters. To assess the influence of zero removal and the different outlier detection methods, we show the R^2 scores for each combination, highlighting our target configuration in bold font.

The reason for this improvement is that request outlier detection is more precise than global outlier detection. For example, consider two requests, x and y, of type T, with processing times of 10 ms and 12 ms, and two requests, a and b, of type M, with processing times of 100 ms and 1000 ms respectively. Global outlier detection would remove both requests a and b, as it neglects the type of request. In contrast, request outlier detection only eliminates request b. Furthermore, zero removal eliminates around 2 million rows from the database, signifying that roughly 10% of our training data consisted of faulty measurements. Yet, the

Table 2. R^2 scores of the estimators when using CMD 0, PR 1 N, PR 2 N as predictor variables, and default hyperparameters. Columns (1)–(4) represent different combinations of zero removal (ZR) and outlier detection method. Column (1) shows the values from Table 1 for easier comparison, depicting the results for no zero removal (ZR) and global outlier detection (GOD). A comparison between Columns (2) and (1) reveals that ZR seemingly has no discernible effect. Columns (3) and (1) show that request outlier detection (ROD) significantly enhances the R^2 score, resulting in an improvement of around 0.2 for all estimators except of Lasso Regression. Column (4) shows the scores for zero removal and request outlier detection. Compared to (3), we observe a marginal improvement for all estimators except Lasso Regression. This underscores that zero removal marginally enhances the R^2 score when implemented alongside request outlier detection. This implies that zero removal boosts the R^2 score specifically when coupled with request outlier detection, albeit with minimal impact due to our specific dataset. The DecisionTree Regression remains the best and Ridge Regression slightly worse, although the difference between their scores has shrunk to 0.053. ElasticNet Regression has significantly improved, but is still far behind Ridge and DecisionTree Regression, with Lasso still being useless.

Estimator	(1) R^2 Score, No ZR, GOD	(2) R^2 Score, ZR, GOD	(3) R^2 Score, No ZR, ROD	**(4) R^2 Score, ZR, ROD**
Ridge Regression	0.537	0.537	0.756	**0.761**
Lasso Regression	−0.000	−0.000	−0.000	**−0.000**
ElasticNet Regression	−0.000	−0.000	0.263	**0.264**
DecisionTree Regression	0.626	0.626	0.811	**0.814**

R^2 score only improves when zero removal is used in conjunction with request outlier detection. This outcome can be attributed to the idiosyncrasies of our specific training data, and thus, we do not generalize that zero removal is only efficacious when paired with a specific outlier detection method.

In our third configuration, we use target encoding of the request type instead of ordinal encoding. Table 3 shows the R^2 scores compared to the previous configurations. The increased score of Ridge Regression can be attributed to the use of target encoding instead of ordinal encoding for the nominal data of request types. Ordinal encoding can introduce an artificial order to the request types, which might lead to a distortion of the model parameter estimated by the Ridge Regression.

Table 4 shows the R^2 scores of our final configuration: we activate automatic hyperparameter optimization for both ordinal and target encoding. We observe that, with this configuration, every estimator has a better score than the best baseline score in Table 1.

To summarize the improvements presented in this paper, cleaning the training data with a better outlier detection method and also removing zero values have together the biggest impact on the prediction quality of the model. Automatic hyperparameter optimization significantly improves the scores of ElasticNet and Lasso Regression, but only slightly improves Ridge and DecisionTree Regression. Target encoding improves Ridge Regression, but has little to no impact on the scores of the other examined estimators.

Table 3. R^2 Scores of the Estimators when using CMD T, PR 1 N, PR 2 N as predictor variables, request outlier detection, zero removal, and default hyperparameters. Compared to Table 2 we use target encoding on the request type instead of ordinal encoding with the result that the R^2 scores of ElasticNet and Lasso Regression are unchanged compared to the previous configuration. The score of the DecisionTree Regression is very slightly lower by 0.004 and Ridge Regression increased by 0.031. The gap between their scores has shrunk further to 0.018.

Estimator	R^2 Score
Ridge Regression	0.792
Lasso Regression	−0.000
ElasticNet Regression	0.264
DecisionTree Regression	0.810

Table 4. R^2 Scores of the Estimators when using CMD O or CMD T, PR 1 N, PR 2 N as predictor variables, request outlier detection, zero removal, and optimized hyperparameters. Compared to Tables 2 and 3 we activate automatic hyperparameter optimization and observe a significant improvement for ElasticNet and Lasso Regression with an improvement of 0.484 for ElasticNet and 0.676 for Lasso Regression for both encodings. The Ridge Regression remained the same. The DecisionTree Regression slightly improved by 0.006 for ordinal encoding and 0.001 for target encoding.

Estimator	R^2 Score CMD O	R^2 Score CMD T
Ridge Regression	0.761	0.792
Lasso Regression	0.676	0.676
ElasticNet Regression	0.748	0.748
DecisionTree Regression	0.820	0.821

The enhancements introduced in this paper improve the R^2 score of the Ridge Regression by 47% and the R^2 score of the DecisionTree Regression by 31%. ElasticNet and Lasso Regression initially had a score of 0, but with our improvements they have a higher score than the best baseline score of 0.626.

5 Experiments: Alarm System's Saturation Point

This section describes experiments aimed at determining the saturation point of our legacy alarm system using the prediction models described in the previous section. We rely on the testing infrastructure developed in our previous work [21,22], based on the popular load testing tool *Locust* [5], together with our own supplementary *shell* and *Python* scripts for automating and visualizing the results. Our testing infrastructure implements the Load Tester, Simulator, and Requirements Checker components of the RAST approach shown in Fig. 1. We observed a performance issue with the initial version of the Simulator component: when approaching the system saturation point, the CPU core

running the simulator process became fully utilized. As a result, the simulation results could not accurately represent the real system, despite having a high R^2 score for the prediction model. The two main reasons for the high CPU utilization are caused by the simulator implementation in Python. First, Python in combination with CPython as the popular Python runtime is an interpreted language and, thus, inherently slower compared to compiled languages. Second, Python's Global Interpreter Lock (GIL) of CPython prevents a Python process from using multiple CPU cores, even when utilizing multiple threads on a multi-core CPU [2]. Therefore, in this paper we re-implement the simulator in the compiled language Kotlin, and we employ the popular *Ktor* library [12] which utilizes multiple CPU cores, thus, solving the performance issues.

To ensure consistent comparison between our new findings and those reported in our previous work [22], we perform our experiments twice - once utilizing the prediction model (PM1) obtained with the initial version of RAST, and then employing the enhanced prediction model (PM2) presented in this paper. We use the same hardware as in our initial study: an HP Proliant dl380 G7 server equipped with two Intel Xeon X5690 processors with 3.46 GHz. We conduct all of our experiments within a virtual machine (VM) that has 8 virtual CPU cores with 16 GB of RAM. We run the Simulator and the Load Tester on the same computer. We also use the popular *mininet* tool [13] for simulating network bandwidth and latency. Our *mininet* topology (hosts, switches, and links) and the link parameters (bandwidth, delay, packet loss, jitter) reflect the properties of our production alarm system. The process of experiment running on our test infrastructure is encapsulated in a single *shell* script that uses *mininet's* built-in functions to establish a *mininet* topology and launch the Simulator and Load Tester components. We allow free access to our source code, *mininet* configuration, and server's parameters GitHub [20].

The goal of our load test is to evaluate the number of alarm devices that can be handled by the system while meeting the real-time requirements under the production workload. The workload generated by the Load Tester has two parts: the system workload generated by alarm devices sending alarm messages and the background workload generated by the other programs running at the same time. Our load tester uses the workload pattern extracted from the production system to generate the background workload; in our case study, it sends 70 requests per second that are randomly selected from 349 different request types. The system workload is gradually increased by simulating the alarm devices, each sending a request every 20 s, and increasing their amount over time, until we reach the critical workload, such that Requirements Checker of RAST reports that the required response time limit is exceeded. In this manner, we determine the saturation point of the system.

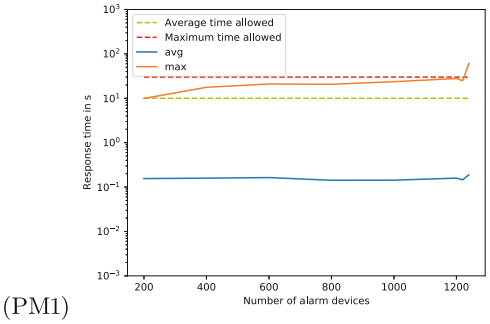
The Load Tester component distributes the system workload among three *Locust* workers, each running as a separate Python process. We experimentally choose the number of workers to be three, with the goal to utilize only as few CPU cores for the experiment as necessary. In our setup, the system workload, the background workload, and Simulator utilize five CPU cores, leaving three

remaining cores for *mininet* and other system processes. The system workload runs for 10 min and then the Load Tester reports the measurements to Requirements Checker. The time of 10 min proves to suffice for each AD to send 30 requests during the experiment. We observe that running the test longer shows no significant influence on the results. If the real-time requirements are met, i.e., the mean and the maximum response times are within the standard requirements described in Sect. 3, Load Tester increases the amount of ADs by 200 (the initial number is 200 too) and repeats the system workload run. The experiment ends once Requirements Checker reports the saturation point. Once the saturation point is found, the Load Tester component goes back to the previous amount of ADs, and then it increases the amount of ADs by 20 in order to get finer-grained results.

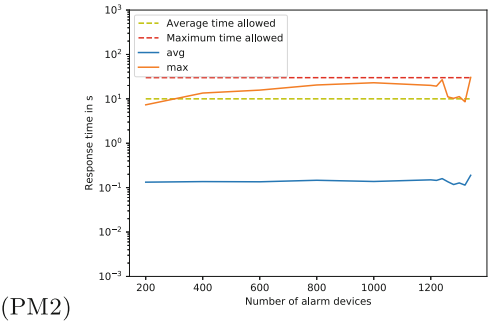
Figure 4 shows the experimental results for our both prediction models, i.e., PM1 (previous) and PM2 (new). The top diagram shows the results for PM1, the bottom diagram for PM2. In each load test executed, the mean and maximum response times are shown as a blue and an orange curve, respectively. We observe that both models yield similar results and the system's saturation point lies around 1240–1340 ADs, depending on the model used. The maximum response time increases quickly and not monotonically. In our experiments, we observe that this behavior happens when several hundred ADs simultaneously send requests within a time frame of less than a second. This behavior is hard to reproduce consistently due to the non-deterministic scheduling of hundreds of ADs and the short time frame in which it occurs. The low average response time suggests that these situations are rare events in the experiment. To mitigate this issue, we repeat the experiment for each model nine times and calculate the average values of all reported average and maximum response times. After nine repetitions, we do not observe any changes in the found saturation point.

Our results show that, although PM2 has a significantly higher R^2 score than PM1, the difference in the measured saturation point is minimal in our experiment. We expect this difference to become more noticeable with a higher number of simulated ADs.

Our study also shows that the value of PR 2 N, which represents the amount of parallel requests that ended while the current request was processed, fluctuates significantly, even for the same number of ADs. These fluctuations are evident in the non-monotonic behavior of the maximum response time shown in Fig. 4, where the maximum response time can be lower for a greater number of ADs. Although our prediction variables produce a high R^2 score, these fluctuations suggest that the adopted prediction variables are still insufficient. Our future research will include other commonly used prediction variables, such as the average amount of requests per sec, to improve the model's accuracy.



(PM1)



(PM2)

Fig. 4. Average and maximum response times depending on the number of ADs for both prediction models. Both models yield similar results, with an average response time (blue curve) between 0.1 s and 0.2 s. The maximum response time (orange curve) exceeds the allowed maximum response time at 1,240 ADs for PM1, with the response time of 61.151 s, and 1,340 ADs for PM2, with the response time of 30.379 s. (Color figure online)

6 Conclusion and Future Work

In this paper, we develop and implement several improvements of RAST – our approach to performance evaluation of legacy real-time systems by combining Regression Analysis, Simulation, and load Testing. These improvements enable training an adequate prediction model for a specific System Under Evaluation. We apply our enhancements to a commercial legacy alarm system which runs already in a production environment and poses strict real-time requirements. We evaluate various RAST configuration settings and report how they affect the model's R^2 score. We demonstrate significantly improved accuracy of performance evaluation due to our new developments.

The results from our prediction model generation indicate a significant improvement of accuracy over our initial study, with R^2 scores increasing for all tested models. For instance, the Ridge Regression improved from 0.537 to 0.792, while the Decision Tree Regression rises from 0.626 to 0.821. Additionally, both Elastic Net and Lasso Regression started out at 0 but were then

increased to respective values of 0.748 and 0.676. Notably, the performance of each model surpassed the highest-performing model from our initial study, which was 0.626. Overall, these findings demonstrate the effectiveness of our proposed methodology.

Our experimental results show that, under usual operating conditions, our evaluated alarm system can successfully handle the workload of up to 1340 Alarm Devices (ADs) while meeting the real-time requirements of the EN 50136 standard. Thus, the company's long-term goal of managing 25000 modern ADs is currently at risk. However, we are now still at an early development stage of our RAST approach: in particular, the current model performs predictions based on only three predictor variables, which leads to fluctuations in the measured response time. Thus, the saturation point obtained with our current prediction model may still not represent the real system accurately enough, despite a high R^2 score.

In our future work, we will extract more predictor variables, like the average amount of requests/sec, from the log files, in order to improve the prediction model. Also, we plan to implement the RAST approach for different (open-source) software systems, in order to estimate its validity for other types of systems and thereby allow for better reproducibility of our experimental results.

References

1. Aichernig, B.K., et al.: Learning and statistical model checking of system response times. Softw. Qual. J. **27**(2), 757–795 (2019)
2. Beazley, D.: Understanding the Python GIL. In: PyCON Python Conference (2010)
3. Belete, D.M., Huchaiah, M.D.: Grid search in hyperparameter optimization of machine learning models for prediction of HIV/AIDS test results. Int. J. Comput. Appl. **44**(9), 875–886 (2022). https://doi.org/10.1080/1206212X.2021.1974663
4. Brownlee, J.: How to remove outliers for machine learning (2020). https://machinelearningmastery.com/how-to-use-statistics-to-identify-outliers-in-data/
5. Byström, C., et al.: Locust. https://docs.locust.io/en/stable/what-is-locust.html
6. Courageux-Sudan, C., Orgerie, A.C., Quinson, M.: Automated performance prediction of microservice applications using simulation. In: MASCOTS 2021, pp. 1–8 (2021). https://doi.org/10.1109/MASCOTS53633.2021.9614260
7. DIN EN 50136-1:2012-08: Alarm systems - alarm transmission systems and equipment - part 1: General requirements for alarm transmission systems (2012)
8. Grohmann, J., et al.: Monitorless: predicting performance degradation in cloud applications with machine learning. In: Middleware 2019, pp. 149–162. ACM (2019). https://doi.org/10.1145/3361525.3361543
9. GS. https://www.gselectronic.com/
10. Huang, J., et al.: An empirical analysis of data preprocessing for machine learning-based software cost estimation. Inf. Softw. Technol. **67**, 108–127 (2015). https://doi.org/10.1016/j.infsof.2015.07.004
11. ISO/IEC: ISO/IEC 2382:2015: Information technology - Vocabulary. Technical report, ISO (2015)
12. JetBrains: Ktor (2022). https://github.com/ktorio/ktor

13. Keti, F., Askar, S.: Emulation of software defined networks using mininet in different simulation environments. In: ISMS 2015, pp. 205–210 (2015). https://doi.org/10.1109/ISMS.2015.46
14. Matthiesen, S., Bjørn, P.: Why replacing legacy systems is so hard in global software development: an information infrastructure perspective. In: CSCW 2015, pp. 876–890. ACM (2015)
15. Okanović, D., Vidaković, M.: Software performance prediction using linear regression. In: Proceedings of the 2nd International Conference on Information Society Technology and Management, pp. 60–64. Citeseer (2012)
16. Pedregosa, F., et al.: Scikit-learn: machine learning in Python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
17. Potdar, K., et al.: A comparative study of categorical variable encoding techniques for neural network classifiers. Int. J. Comput. Appl. **175**(4), 7–9 (2017). https://doi.org/10.5120/ijca2017915495
18. Regression analysis essentials for machine learning. https://www.sthda.com/english/articles/40-regression-analysis/
19. Scikit-learn Development Team: Model selection and evaluation. https://scikit-learn.org/stable/model_selection.html
20. Tomak, J.: Performance Testing Infrastructure (2022). https://github.com/jtpgames/Locust_Scripts
21. Tomak, J., Gorlatch, S.: Measuring performance of fault management in a legacy system: an alarm system study. In: Calzarossa, M.C., Gelenbe, E., Grochla, K., Lent, R., Czachórski, T. (eds.) MASCOTS 2020. LNCS, vol. 12527, pp. 129–146. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-68110-4_9
22. Tomak, J., Gorlatch, S.: RAST: evaluating performance of a legacy system using regression analysis and simulation. In: MASCOTS 2022, pp. 49–56 (2022). https://doi.org/10.1109/MASCOTS56607.2022.00015