






Enhancing Infrastructure Observability: Machine Learning for Proactive Monitoring and Anomaly Detection

Darlan Noetzold   [Instituto Federal de Educação, Ciência e Tecnologia Sul-Rio-Grandense | darlan.noetzold@gmail.com]

Anubis G. D. M. Rossetto  [Instituto Federal de Educação, Ciência e Tecnologia Sul-Rio-Grandense | anubisrossetto@ifsul.edu.br]

Valderi R. Q. Leithardt   [Instituto Universitário de Lisboa (ISCTE-IUL), ISTAR, Lisboa, Portugal | valderi.leithardt@iscte-iul.pt]

Humberto J. de M. Costa  [Instituto Federal de Educação, Ciência e Tecnologia Sul-Rio-Grandense | humberto.costa@osorio.ifrs.edu.br]

 Federal Institute of Education, Science and Technology Sul-rio-grandense, Passo Fundo, Rio Grande do Sul, RS, 99064-440, Brazil.

Instituto Universitário de Lisboa (ISCTE-IUL), Av. das Forças Armadas, 1649-026, Lisboa, Portugal.

Received: 23 May 2024 • Accepted: 21 August 2024 • Published: 28 October 2024

Abstract This study addresses the critical challenge of proactive anomaly detection and efficient resource management in infrastructure observability. Introducing an innovative approach to infrastructure monitoring, this work integrates machine learning models into observability platforms to enhance real-time monitoring precision. Employing a microservices architecture, the proposed system facilitates swift and proactive anomaly detection, addressing the limitations of traditional monitoring methods that often fail to predict potential issues before they escalate. The core of this system lies in its predictive models that utilize Random Forest, Gradient Boosting, and Support Vector Machine algorithms to forecast crucial metric behaviors, such as CPU usage and memory allocation. The empirical results underscore the system's efficacy, with the GradientBoostingRegressor model achieving an R^2 score of 0.86 for predicting request rates, and the RandomForestRegressor model significantly reducing the Mean Squared Error by 2.06% for memory usage predictions compared to traditional monitoring methods. These findings not only demonstrate the potential of machine learning in enhancing observability but also pave the way for more resilient and adaptive infrastructure management.

Keywords: Machine Learning, Infrastructure Monitoring, Anomaly Detection, Proactive Maintenance

1 Introduction

In the era of digital transformation, software systems have become fundamentally complex [dos Santos *et al.*, 2021], driven by distributed architectures such as microservices, which bring with them significant challenges for monitoring and ensuring quality of service [Borré *et al.*, 2023]. Observability, unlike traditional monitoring, is conceptualized as the ability to infer the internal state of a system from its external data, such as logs, metrics, and traces, making it easier to diagnose problems and understand the behavior of the system in production.

Given the complexity of modern systems and the vast amount of data generated [Surek *et al.*, 2023], it is crucial not only to collect but also to analyze and interpret this data efficiently. In this context, predictive approaches, supported by machine learning techniques [Yamasaki *et al.*, 2024], present as a natural evolution of observability practices, enabling the proactive identification of potential failures and performance problems before they negatively impact end users or business operations [Corso *et al.*, 2023].

The main contributions of this paper are threefold:

- **Integration of Predictive Models:** We introduce an innovative integration of predictive models with observabil-

ity platforms, which enhances the precision of real-time monitoring and proactive anomaly detection.

- **Improved Prediction Accuracy:** We demonstrate significant improvements in prediction accuracy using advanced machine learning algorithms such as Random Forest, Gradient Boosting, and Support Vector Machines.
- **Scalable Framework:** Our approach provides a novel framework for implementing machine learning models within microservices architectures, facilitating scalability and modularity in infrastructure management.

These contributions distinguish our work from existing methods by offering improved detection capabilities and seamless integration with modern observability tools.

This paper proposes an advanced observability approach that integrates predictive models to anticipate the future state of monitored metrics, with the aim of detecting anomalies early and optimizing interventions. Based on the theoretical foundations of observability and machine learning, the research explores the effectiveness of this integration in a case study applied to a production environment, comparing its performance with traditional monitoring methods.

The methodology employed involves the collection and analysis of metrics from real systems, followed by the appli-

cation of predictive models to establish predictions about the behavior of these metrics. The results are evaluated based on standard performance metrics for machine learning models, such as mean square error (MSE) and coefficient of determination (R^2), providing a quantitative basis for comparing predictive and traditional approaches [Tarek *et al.*, 2023].

The literature surveyed indicates an improvement in the capacity for early detection of problems and in the efficiency of operational interventions [Stefenon *et al.*, 2023d], validating the hypothesis that the integration of predictive techniques into observability can make a significant contribution to the management of complex systems. However, the importance of additional considerations, such as the interpretability of the models and the impact of latency in data collection, for the practical implementation of this approach is highlighted [Singh *et al.*, 2023].

The paper is organized as follows: Section II discusses the tools and technologies used, highlighting the importance of each technological selection for the execution of the project. Section III discusses related work, providing an overview of existing solutions and highlighting the differences between these approaches. Section IV details the methodology adopted to develop and integrate predictive models with system observability, including data collection and preparation, development of Machine Learning models, and model evaluation. Section V focuses on implementation, explaining the process of data collection, preparation, data normalization for modeling, model training, and the implementation of the prediction Application Programming Interface (API). The results obtained are presented in Section VI, where the efficiency of the microservices architecture is analyzed and the results of the predictive models are discussed through the analysis of accuracy graphs. The Conclusions and future work in Section VII summarize the contributions of the study, the results obtained, and suggest directions for future work.

2 Theoretical Foundations

This section presents a detailed exploration of the technological and methodological bases that support the application. The initial section discusses the fundamental tools and technologies that make up the application's architecture, followed by a description of the prediction models in two distinct layers: the first focused on predicting the future values of the metrics and the second on determining alerts. Finally, the methods used to evaluate the performance of the models will be detailed.

Prediction Models and Observability Concepts Observability, in modern software systems, transcends traditional monitoring by enabling the inference of a system's internal state from its external outputs [Corso *et al.*, 2023]. It leverages data such as logs, metrics, and traces to diagnose problems and understand system behaviors in production environments. Prediction models play a crucial role in enhancing observability by analyzing historical data to forecast future states of system metrics, thereby allowing for proactive identification and resolution of potential issues before they im-

part end users [Stefenon *et al.*, 2023c]. The integration of machine learning techniques into observability frameworks facilitates this predictive capability, making it possible to anticipate anomalies and optimize system performance in real-time [Yamasaki *et al.*, 2024].

Theoretical justifications for this integration can be found in control theory and time-series analysis. Control theory provides a foundation for understanding how systems can be monitored and controlled based on their output data [da Silva *et al.*, 2024]. Time-series analysis offers methods to analyze temporal data, making it possible to predict future values based on past observations [Ribeiro *et al.*, 2024]. These theories support the use of machine learning algorithms, which can model complex, non-linear relationships in data, providing accurate predictions that enhance observability practices.

Tools and Technologies The application's architecture was built based on modern software development principles, emphasizing scalability, modularity, and efficiency. The main technologies adopted include:

- **Spring Framework:** Used to create a robust and flexible base for the development of microservices, allowing the injection of dependencies and facilitating the management of application components [Webb *et al.*, 2013].
- **Redis:** Used as a caching system to improve the performance of read operations, reducing the load on the database and speeding up the retrieval of frequently accessed data [Da Silva and Tavares, 2015].
- **RabbitMQ:** Adopted for communication between microservices via messages, guaranteeing reliable delivery of information and allowing processes to be decoupled [Dossot, 2014].
- **Cython:** Used to increase the performance of data processing and prediction algorithms by compiling critical parts of the code to C, resulting in faster executions [Behnel *et al.*, 2010].
- **Scikit-learn (Sklearn):** One of Python's most popular and versatile machine learning libraries, selected for its vast collection of modeling algorithms for both regression and classification, as well as features for data preprocessing, model selection, and evaluation. Its use was essential for building and optimizing our predictive models [Kramer and Kramer, 2016].
- **Pandas:** High-performance, easy-to-use data manipulation, and analysis library, enabling efficient manipulation of large data sets, cleaning, transformation, and analysis. It was crucial for preparing and exploring the data before applying the predictive models [McKinney, 2018].
- **PostgreSQL:** Robust relational database management system with SQL support, selected for its reliability, advanced features (such as JSON support), and compatibility with large volumes of data. Its adoption guarantees the integrity and security of the data handled by the application [Milani, 2008].

Prediction Models The application uses two levels of prediction models, each with specific objectives:

1. First Layer - Value Prediction: Uses regressive models to predict the future values of monitored metrics. Selected models include:

- **RandomForestRegressor**: For its ability to model non-linear relationships without the need for extensive data pre-processing [Rodriguez-Galiano *et al.*, 2015].
- **GradientBoostingRegressor**: Because of its effectiveness in reducing bias and variance by combining multiple weak decision trees into a strong model [Elango *et al.*, 2022].
- **Support Vector Regression (SVR)**: Because of its robustness in dealing with high dimensions and its efficiency in finding the optimal regression hyperplane [Zhang and O'Donnell, 2020].

2. Second Layer - Alert Prediction: Focuses on predicting the need to issue alerts (`is_alert`) based on trends in metrics. Classification models include:

- **RandomForestClassifier**: selected for its ability to deal with unbalanced datasets and for its interpretability [Liu *et al.*, 2012].
- **GradientBoostingClassifier**: For its accuracy and ability to deal with overfitting through gradient optimization [Chakrabarty *et al.*, 2019].
- **LogisticRegression**: Due to its simplicity and effectiveness in binary classification problems, as well as the ease with which the results can be interpreted [Christodoulou *et al.*, 2019].

To optimize the models, we used `RandomizedSearchCV` and `BayesSearchCV`, which allow an efficient search for the best hyperparameters. The former performs random searches within a defined space, ideal for broad exploration with lower computational costs. The second adopts a more focused and efficient Bayesian optimization approach, especially useful when the computational cost of the models is high.

Before training the models, the data is normalized using `StandardScaler` from the Scikit-learn library. This normalization process standardizes the features of the data by subtracting the mean and scaling to the unit variance [Aldi *et al.*, 2023]. This step is crucial for models that are sensitive to the scale of the data, such as SVM, and helps improve convergence during training.

Evaluation Methods The evaluation of prediction models is essential to determine their effectiveness in predicting future values. In this sense, metrics such as MSE (Mean Squared Error), MAE (Mean Absolute Error), R^2 (R Squared) and Explained Variance play a crucial role [Stefenon *et al.*, 2024]. In the current work, these metrics were applied using the Scikit-learn library, which already has optimized implementations of all the calculations that will be presented.

The MSE is calculated using the equation:

$$MSE = \frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2. \quad (1)$$

The MSE strongly penalizes large deviations between the model's predictions (\hat{y}_i) and the actual values (y_i).

In the context of the algorithm, the MSE is calculated using the function `mean_squared_error(y_test, predictions)`, providing a measure of the dispersion of the squared errors of the predictions [Ribeiro *et al.*, 2024].

The MAE, in turn, is obtained from the formula:

$$MAE = \frac{1}{n} \sum_{i=1}^n |\hat{y}_i - y_i|, \quad (2)$$

representing the average of the absolute differences between the predictions and the actual values. In the algorithm, the MAE is calculated using `mean_absolute_error(y_test, predictions)`, providing a more direct measure of the average error of the predictions, which is less sensitive to extreme values.

The R^2 , or coefficient of determination, is expressed by the formula:

$$R^2 = 1 - \frac{\sum_{i=1}^n (\hat{y}_i - y_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2}, \quad (3)$$

where \bar{y} denotes the mean of the actual values [da Silva *et al.*, 2024]. This metric indicates the proportion of the variance in the data that is explained by the model. In the algorithm, we use `r2_score(y_test, predictions)` to calculate the R^2 , providing a measure of how well the model fits the observed data.

Finally, the Explained Variance, determined by the formula:

$$\text{Explained Variance} = 1 - \frac{\text{Var}(y - \hat{y})}{\text{Var}(y)}, \quad (4)$$

where $\text{Var}(y - \hat{y})$ represents the variance of the residual errors, measuring the proportion of the variance of the data captured by the model. In the algorithm, the Explained Variance is calculated with `explained_variance_score(y_test, predictions)`, providing insights into how the model captures the variations in the data.

To summarize the difference between the metrics, recognize good results, and understand what each accuracy evaluates, you can look at Figure 1, which explains these points. These metrics are used to evaluate the performance of prediction models, allowing the performance of models to be compared and the selection of the one that offers the best predictions for real-time monitoring of system metrics.

Metric	Description	Best Value	Worst Value
MSE	Penalizes deviations between predictions and actual values.	0	∞
MAE	Provides a direct measure of the average prediction error.	0	∞
R^2	Proportion of variance in the data explained by the model.	1	$-\infty$
Explained Variance	Proportion of variance in the data captured by the model.	1	$-\infty$

Table 1. Evaluation of prediction metrics.

Tensor Processing Unit The Tensor Processing Unit (TPU) is an ASIC (Application-Specific Integrated Circuit) developed by Google specifically to speed up tasks related

to machine learning [Junior *et al.*, 2022]. It is designed to handle multidimensional matrix operations, which are common in neural network algorithms and other deep learning models [Jouppi *et al.*, 2018]. However, in this study, TPUs were used primarily as part of the Google Colab infrastructure to facilitate faster data processing and model training. Given that the models employed (Random Forest, Gradient Boosting, and Support Vector Machines) are not deep learning models, the impact of using TPUs was minimal. The computational efficiency required for these algorithms did not necessitate the advanced capabilities of TPUs, and the primary performance improvements were achieved through algorithmic optimizations and the microservices architecture implemented.

3 Related Work

The observability of systems and applications is a crucial field for maintaining the reliability, availability, and performance of digital services [Stefenon *et al.*, 2023a]. Tools such as Zabbix, Dynatrace, Prometheus, Grafana, and Nagios represent fundamental monitoring solutions, offering real-time insights into the state of systems and enabling rapid detection and resolution of problems.

Zabbix is valued for its ability to monitor thousands of metrics collected from servers, network devices, and applications, as well as providing alerts and detailed visualizations [Zabbix, 2024]. Dynatrace employs artificial intelligence to provide complete observability of applications, services, and infrastructures, excelling in the automatic analysis of anomalies [Ahola, 2022; Dynatrace, 2024]. Prometheus is recognized for its robust ability to monitor distributed systems, with a multidimensional data model and a powerful query engine [Turnbull, 2018]. Grafana offers advanced visualization and analysis, allowing the creation of dynamic dashboards from multiple sources [Chakraborty, 2021]. Nagios monitors the health of the IT infrastructure, alerting you to potential problems before they affect critical processes [Barth, 2008].

Despite the strengths of each of these tools, they have limitations, especially in their ability to predict and proactively prevent incidents, as can be seen in Table 2. Current research seeks to complement these existing capabilities by introducing advanced prediction mechanisms that analyze historical trends to predict future states of monitored metrics, aimed not only at detecting existing problems but also at predicting possible incidents. The proposed project in this article includes all the functionalities listed in the table.

Recent studies have further explored the integration of machine learning techniques with observability to enhance the predictive capabilities of monitoring tools. Hao *et al.* [2022] conducted a nonlinear observability analysis of multi-robot cooperative localization, showcasing advanced methods in the observability domain. Vos *et al.* [2023] provided a systematic literature review on generalizable machine learning models for stress monitoring from wearable devices, highlighting the challenges and opportunities in this field. Min *et al.* [2021] developed a stochastic machine learning approach to enhance observability in automated smart grids, focusing on the positioning of micro-synchrophasor units.

These recent works illustrate the growing interest and progress in combining machine learning with observability practices. However, there remains a significant gap in the literature regarding the practical application of these techniques in real-world environments and their integration into existing monitoring tools. Most existing studies are either theoretical or conducted in simulated environments, lacking implementation in actual production systems. Moreover, the specific integration of these machine learning models with established monitoring tools like Zabbix or Prometheus is not extensively covered.

The structure based on microservices, the use of technologies such as Spring, Redis for caching, and RabbitMQ for messaging, contributes to the scalability, modularity, and efficiency of the proposed solution. Integrating this predictive capability into existing monitoring tools, such as a plugin or embedded component, offers a valuable extension to their functionality, reinforcing the importance of prediction and prevention in proactive systems management [De Souza *et al.*, 2020; Leithardt *et al.*, 2020].

Unlike traditional monitoring solutions, which focus on detecting and warning of conditions that have already manifested, this research emphasizes prediction and prevention as key strategies for proactive system management, taking the practice of observability to a new level of effectiveness.

4 Methodology

The methodology applied covers the collection and preparation of data, the training of predictive models via algorithms obtained from the Scikit Learn library, and the evaluation of these models using metrics such as MSE and R^2 , culminating in their integration into monitoring systems via a microservices architecture. This process is supported by key technologies, including Spring Framework, Redis, RabbitMQ, and the use of TPUs for training acceleration, aimed at optimizing observability in IT infrastructures.

4.1 Planning

The design of the application and the selection of the technologies incorporated were based on a planning scheme structured in several stages. First, there was the data acquisition phase, followed by the pre-processing stage, in which the data was cleaned and prepared for analysis. Subsequently, machine learning algorithms were trained using data sets to model complex patterns. This stage was followed by evaluating the performance of the models and applying statistical and computational metrics to ensure accuracy and predictive effectiveness. Finally, the integration phase consolidated the predictive models developed with the existing monitoring platforms. More details of each stage are presented below.

Data Collection The data collection phase involves acquiring system and application metrics through instrumentation and monitoring [Moreno *et al.*, 2024]. Standard metrics such as CPU usage, memory, disk space, request response time,

Feature	[Olups, 2016]	[Ahola, 2022]	[Chakraborty, 2021]	[Barth, 2008]	[Turnbull, 2018]	[Hao et al., 2022]	[Vos et al., 2023]
Metric Monitoring	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Anomaly Analysis	No	Yes	No	No	No	Yes	Yes
Value Prediction	No	No	No	No	No	No	Yes
Alert Forecasting	No	No	No	No	No	No	No
ML Integration	No	Yes	No	No	No	Yes	Yes
Scalability	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Modularity	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Table 2. Comparison between monitoring tools and the proposed project.

and error rate are continuously collected and stored in a PostgreSQL database for future analysis. The datasets used in this study are composed of metrics collected from real-world production environments. The sources include server logs, application performance monitoring tools, and infrastructure monitoring systems. The data spans a period of six months, capturing a wide range of operational conditions and performance states. The dataset consists of approximately 10 million records, each containing multiple features relevant to system performance and health.

- **Sources:** Server logs, Application performance monitoring tools (e.g., Java Monitor Control, Java Flight Recorder) and Infrastructure monitoring systems (e.g., Prometheus, Grafana);
- **Size:** Approximately 10 million records;
- **Features:** CPU usage (%), Memory usage (MB), Disk space usage (GB), Request response time (ms), Error rate (%), Network throughput (Mbps), Database query times (ms) and Application-specific metrics (e.g., number of active users, transaction counts);
- **Labels:** Anomaly detection (binary label indicating normal or anomalous state) and Predicted metric values (continuous values for future state prediction).

Data Preparation The collected data goes through a preparation process before modeling, which includes several steps to ensure the quality and suitability of the data for machine learning models:

- **Data Cleaning:** Removing outliers, handling missing values through imputation techniques (e.g., mean substitution), and filtering out irrelevant data points to ensure data integrity.
- **Normalization:** Using StandardScaler from the Scikit-learn library to standardize features by subtracting the mean and scaling to unit variance, optimizing the performance of machine learning models [Stefenon et al., 2023b].
- **Feature Engineering:** Transforming the time series data into a format suitable for predictive modeling. This includes creating additional features from existing data, such as rolling averages, lagged variables, and interaction terms, to enhance model performance.
- **Data Splitting:** Dividing the dataset into training, validation, and test sets to ensure robust model evaluation

and prevent overfitting.

Development of Machine Learning Models Two sets of Machine Learning models are developed: the first to predict the future values of the metrics (first layer) and the second to predict the occurrence of alerts (second layer). Models such as RandomForestRegressor, GradientBoostingRegressor and SVR are used in the first layer, while RandomForestClassifier, GradientBoostingClassifier, and LogisticRegression are employed in the second layer. Hyperparameter search techniques, including RandomizedSearchCV for random exploration of the hyperparameter space and BayesSearchCV for a more targeted and efficient search, are applied to model selection and optimization.

Model evaluation Models are evaluated using standard performance metrics, including Mean Squared Error (MSE), Mean Absolute Error (MAE), R^2 (coefficient of determination) and Explained Variance for regressor models. This phase allows the selection of the most accurate and robust models for implementation in the application [Klaar et al., 2023].

Integration and Deployment The selected models are integrated with an API and an interface for testing that is structured as a set of microservices. The application is able to process metrics in real-time, apply the predictive models and generate predictive alerts, accessible via a user interface or sendable to existing monitoring systems via API or plugins developed for this purpose.

4.2 Implementation

To develop the architecture and the application itself, it was modularized, making it easier for third parties to maintain and understand. The "Data Collection and Preparation" section discusses how data is acquired and prepared for analysis. Next, in "Solution Architecture", the configuration of microservices, the use of Spring Boot in the API Gateway, and caching and messaging technologies are detailed. "Normalization and Preparation of Data for Modelling" examines the transformation of data for training models. The subsections "Training the First Layer of Models" and "Training the Second Layer of Models" describe, respectively, the process of

training the regression and classification models, focusing on the selection and optimization of hyperparameters. The "Implementation of the Prediction API" section illustrates how the trained models are applied to make real-time predictions. Finally, the "User Interface" reveals how the React application allows you to interact with the system, offering visualization of the metrics and access to the predictive models.

The project was implemented on a home server equipped with 1 TB of SSD, 16 GB of DDR4 RAM and an Intel Core i3 6100 processor. The hardware supports an Ubuntu Server environment, enabling the deployment and execution of the necessary tests.

4.2.1 Architecture Overview

The architecture diagram, as shown in Figure 1, illustrates the comprehensive design and interactions between various components of the monitoring application. Each module within this architecture has a defined role and communicates with other modules to ensure the smooth functioning of the system.

External API

- **Role:** The API being monitored.
- **Communication:** The API Gateway continuously monitors the External API, collecting current metrics to predict next values and determine if these values will trigger alerts.

API Collector

- **Role:** Collects metrics from the External API.
- **Communication:** Sends the collected metrics to the PostgreSQL Database (Metrics Storage).

PostgreSQL Database (Metrics Storage)

- **Role:** Stores the collected metrics from the API Collector.
- **Communication:**
 - Receives metrics from the API Collector.
 - Provides datasets to the Python scripts for training the first and second layers of models.

Python Scripts

- **First Layer Script:**
 - **Role:** Trains the first layer of models using datasets from the PostgreSQL Database.
 - **Execution:** Runs nightly to update the models with recent data.
 - **Communication:**
 - * Fetches datasets from the PostgreSQL Database.
 - * Saves trained models to a folder on the Linux server.
 - * Sends accuracy metrics to the API Gateway.
- **Second Layer Script:**

- **Role:** Trains the second layer of models using datasets from the PostgreSQL Database.
- **Execution:** Runs nightly to update the models with recent data.
- **Communication:**
 - * Fetches datasets from the PostgreSQL Database.
 - * Saves trained models to a folder on the Linux server.
 - * Sends accuracy metrics to the API Gateway.

Linux Server Folder

- **Role:** Stores the trained models.
- **Communication:** The Python API retrieves the trained models from this folder for prediction and alert detection.

Python API

- **Role:** Provides endpoints for the API Gateway to send current metrics of the External API.
- **Function:** Uses the trained models from the Linux Server Folder to predict the next values and determine if these values will trigger alerts.
- **Communication:**
 - Receives current metrics from the API Gateway.
 - Retrieves trained models from the Linux Server Folder.
 - Sends predictions and alert statuses back to the API Gateway.

API Gateway

- **Role:** Acts as a central hub for external queries and interactions.
- **Functions:**
 - Monitors the External API for real-time metrics.
 - Sends current metrics to the Python API for prediction and alert evaluation.
 - Stores accuracy metrics and prediction results in another PostgreSQL Database (Accuracy Storage).
 - Exposes endpoints for external queries and integrates with different monitoring applications.
- **Communication:**
 - Continuously communicates with the External API for metrics.
 - Interacts with the Python API for predictions.
 - Sends accuracy and prediction results to the PostgreSQL Database (Accuracy Storage).
 - Provides endpoints for the Front-end for user feedback and testing.

PostgreSQL Database (Accuracy Storage)

- **Role:** Stores accuracy metrics and prediction results.
- **Communication:** Receives data from the API Gateway.

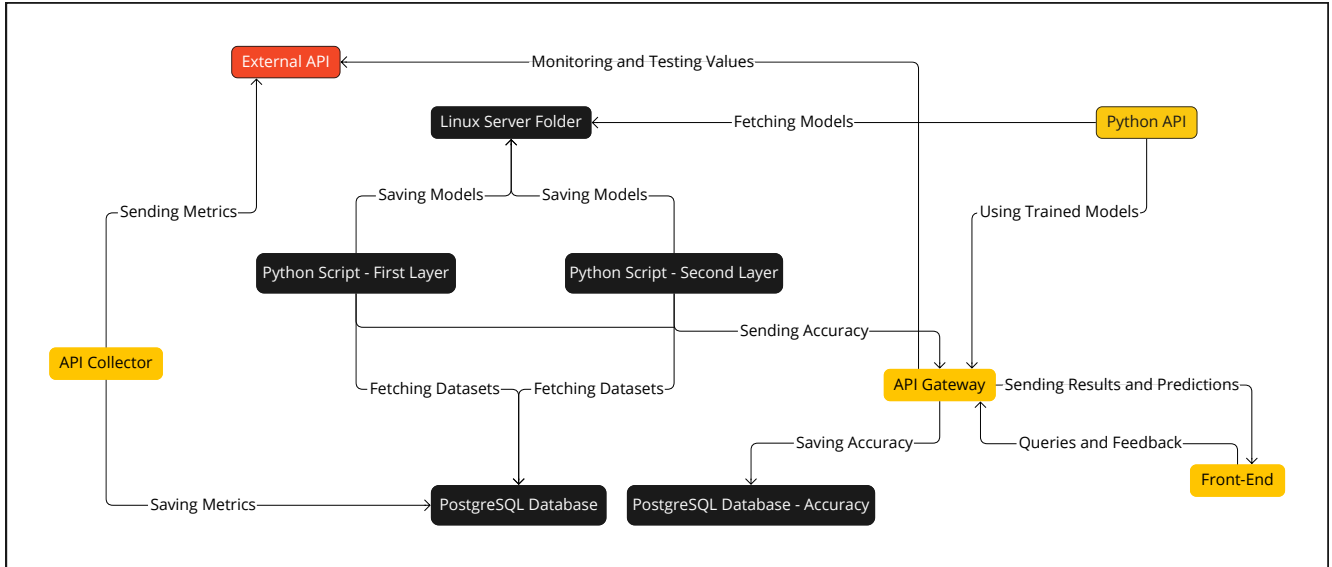


Figure 1. Architecture Diagram of the Monitoring Application

Front-end

- **Role:** Provides a user interface for testing and feedback.
- **Communication:** Interacts with the API Gateway to display alerts and prediction results, providing user feedback.

This architecture ensures a robust and scalable solution for real-time monitoring, predictive analysis, and anomaly detection, leveraging the power of machine learning models integrated into a microservices framework. The application was created to analyze single-applications on single-servers. To add more applications, it would be necessary to duplicate the application (another Docker container for example). This is because models are trained specifically for an application, which may have behaviors that differ from other applications.

Data Collection Module Implemented in Spring Boot, this module captures metrics such as CPU consumption, memory, disk space and HTTP request response time. The data is stored in PostgreSQL and a stored procedure is used to update the `is_alert` flag based on thresholds to detect abnormal conditions (see Listing 1). This procedure begins by resetting the `is_alert` flag for all records in the measurement table, ensuring that only metrics that exceed the defined thresholds are marked as alerts. Next, a loop is executed for each metric stored, where the average (`avg_val`) and standard deviation (`stddev_val`) of the values collected are calculated. Based on these values, a dynamic threshold is defined for each metric (`var_threshold := var_avg_val + var_stddev_val;`), which serves as the criterion for updating the `is_alert` flag.

This stored procedure optimizes database update operations, making the process faster and independent of the application’s logic. By adjusting the thresholds based on the historical behavior of the metrics, the application adapts to changes in the usage pattern of the monitored applications, ensuring relevant alerts and reducing false positives. Integrating this strategy into the database improves efficiency

and allows for more precise resource management, reinforcing the system’s ability to offer real-time insights.

Listing 1: Stored procedure to update the `is_alert` flag.

```

CREATE OR REPLACE PROCEDURE update_alert_flag()
LANGUAGE plpgsql
AS $$
DECLARE
    var_name TEXT;
    var_avg_val DOUBLE PRECISION;
    var_stddev_val DOUBLE PRECISION;
    var_threshold DOUBLE PRECISION;
BEGIN
    -- Reset all alerts
    UPDATE measurement m
    SET is_alert = FALSE;

    -- Loop through each metric to calculate
    --   ↳ threshold and update alert flags
    FOR var_name, var_avg_val, var_stddev_val IN
    SELECT
        mr.name,
        AVG(m.value) AS avg_val,
        STDDEV(m.value) AS stddev_val
    FROM measurement m
    JOIN metric_response mr ON m.
        ↳ metric_response_id = mr.id
    GROUP BY mr.name
    LOOP
        var_threshold := var_avg_val +
            ↳ var_stddev_val; -- Define the
            ↳ threshold

        -- Update the is_alert flag where the
        --   ↳ measurement value exceeds the
        --   ↳ threshold
        UPDATE measurement m
        SET is_alert = TRUE
        FROM metric_response mr
        WHERE m.metric_response_id = mr.id
        AND mr.name = var_name
        AND m.value > var_threshold;
    END LOOP;
END;
$$;
    
```


Data Preparation and Normalization Module Data preparation and normalization is carried out using the Cython script on Listing 2, which makes use of the NumPy library to manipulate numerical arrays, and Scikit-learn’s StandardScaler to normalize the data, ensuring that it has a zero mean and unit variance. This step is crucial to prevent variations in the scale of the data from negatively influencing the performance of Machine Learning models.

Listing 2: Cython script for data preparation and normalization

```
cimport numpy as cnp

def prepare_data(cnp.ndarray[cnp.float64_t, ndim
    ↳ =1] series, int n=10):
    cdef int i
    cdef list X = []
    cdef list y = []
    if series.shape[0] > n:
        for i in range(series.shape[0] - n):
            X.append(series[i:i+n])
            y.append(series[i+n])
    return np.array(X, dtype=np.float64), np.
        ↳ array(y, dtype=np.float64)

def process_metric(filename):
    if filename.endswith(".csv"):
        metric_name = filename[:-4]
        df = pd.read_csv(os.path.join(input_dir,
            ↳ filename), usecols=['
            ↳ measurement_value']).dropna()

    if not df.empty:
        X, y = prepare_data(df['
            ↳ measurement_value'].values.
            ↳ astype(np.float64), n=10)
        X_train, X_test, y_train, y_test =
            ↳ train_test_split(X, y,
            ↳ test_size=0.2, random_state
            ↳ =42)

        scaler = StandardScaler()
        X_train_scaled = scaler.fit_transform
            ↳ (X_train)
        X_test_scaled = scaler.transform(
            ↳ X_test)

        args_list = [(metric_name,
            ↳ X_train_scaled, X_test_scaled,
            ↳ y_train, y_test, model_name,
            ↳ mp) for model_name, mp in
            ↳ models_params.items()]
```

Hyperparameter Fitting Module The models are fine-tuned using RandomizedSearchCV and BayesSearchCV from the Scikit-learn and Scikit-optimize libraries, respectively. RandomizedSearchCV allows the hyperparameters to be explored randomly within a predefined space, while BayesSearchCV uses Gaussian processes to model the hyperparameter space and optimize the models. These techniques are used here to find the best combination of hyperparameters that maximizes the accuracy of predictive models.

Model Training - First Layer Scikit-learn regression models are used to predict the future values of the metrics: RandomForestRegressor, to capture complex non-linear relationships between variables;

GradientBoostingRegressor, which builds models additively using boosting algorithms; and SVR, a model based on support vector machines for regression, which is effective in high-dimensional spaces. Each metric is modeled separately, resulting in a total of 78 different models. The models are serialized and saved in .pkl files using the joblib library for future use. The hyperparameters were adjusted based on detailed empirical tests to optimize the performance of the models, as presented in Table 3 [Noetzold, 2024a]. Finally, all the models were tested again using the Scikit-learn library, which contains the implementations of the metrics presented earlier.

Model	n estimator	max depth	min samples split	min samples leaf	learning rate / C / gamma
Random Forest Regressor	100-200	5-10	2-5	1-4	-
Gradient Boosting Regressor	100-200	3-6	-	-	0.01-0.05
SVR	-	-	-	-	C: 1-100, gamma: scale

Table 3. Tuned hyperparameters for regression models

Initially, the tests were carried out on the aforementioned home server hardware. However, there was a need to improve the training of the regression models on a more robust infrastructure. For this reason, the training of the first layer was transferred to Google Colab Pro, which offers a more powerful cloud infrastructure for machine learning. In this new environment, Google’s TPU V2 architecture was used [Noetzold, 2024b].

Model Training - Second Layer The second layer focuses on classifying metric values as normal or abnormal (is_alert). Classifiers are used such as RandomForestClassifier, which offers good performance and generalization capacity; GradientBoostingClassifier, applying boosting to build strong predictive models; and LogisticRegression, a simple but effective model for binary classification problems. Similar to the first layer, models are trained for each of the 26 metrics, resulting in 78 classifier models. Data normalization and hyperparameter adjustment follow the same process described above, but now with the parameters defined in Table 4. Finally, the models are subjected to the same tests as in the first layer.

Model	n estimators	max depth	learning rate / C
RandomForest Classifier	200-300	15, 20	-
GradientBoosting Classifier	200-300	-	0.1-0.2
LogisticRegression	-	-	10, 100, 1000

Table 4. Adjusted hyperparameters for the classification models

API Python for Querying Models A Python API, developed with the Flask framework, provides two main endpoints: one for predicting the future values of the metrics (using the models in the first layer) and another for classifying the values as alerts (second layer). The API is designed to be lightweight and easy to integrate, providing a RESTful interface for consulting the trained models.

API Gateway The API Gateway, built in Spring Boot, serves as the entry point for requests, directing them to the Python API or directly handling CRUD operations related to model metrics and accuracy. Redis is used for caching, reducing response times for frequently accessed data, and RabbitMQ for decoupling components, facilitating the application's scalability and resilience.

User Interface The user interface is developed in React, creating a SPA (Single Page Application) that consumes the API Gateway endpoints. This interface allows users to view the metrics in real time, test the prediction and classification models, and view the accuracy of the models, offering an interactive and user-friendly experience.

5 Results

The preliminary results obtained, which are detailed in this section, present the feasibility of the application in accordance with the objectives proposed previously. An analysis is made of the four main metrics selected for this study and the efficiency of the proposed architecture in relation to the maintenance of active functionalities is discussed. In addition, the results obtained in the Machine Learning models are detailed, based on the graphs generated by the R^2 and MSE scores (the Explained Variance and MAE scores were similar, which is why they will not be detailed in this paper).

5.1 Selection of metrics for analysis

Of the 26 metrics evaluated, four main ones were selected for in-depth study in this paper: `http_server_requests_active`, `jvm_memory_used`, `jvm_threads_started` and `system_cpu_usage`. These metrics were selected because they represent critical aspects of the performance and health of the monitored applications, and are fundamental for understanding and analyzing their behavior under different operating conditions. The other metrics were not added to this paper simply due to lack of space and so that it does not become repetitive. Therefore, for a detailed examination of the results relative to the other metrics, we direct the reader to the project's GitHub repository [Noetzold, 2024a].

5.2 Microservice Architecture Performance

The chosen microservice architecture demonstrated significant positive outcomes during the testing phase, confirming its suitability for the monitoring application. Throughout the tests, none of the APIs—API Gateway, API Collector, and the Python API utilizing the trained models—showed any

signs of bottlenecks or performance issues. All components maintained a consistent and efficient response to requests.

Specifically, the architecture's performance metrics were impressive:

- **CPU Usage:** All APIs maintained CPU usage below 13% during peak loads.
- **Memory Usage:** Memory consumption was kept below 19%, ensuring there were no memory-related performance issues.

These results highlight the robustness of the microservice architecture, emphasizing its capacity to handle concurrent requests efficiently without compromising performance. This architectural approach has been supported by previous studies, demonstrating that it maintains functionality and performance even under varying demand conditions. The inherent modularity of this architecture facilitates timely interventions and maintenance, reducing service interruptions and ensuring continuous operation. Additionally, the scalability of microservices, as detailed in Noetzold *et al.* [2023], enables the system to effectively manage load fluctuations, which is crucial for the ongoing collection and analysis of data required in this research.

5.3 User Interface Results

The front-end interface of the monitoring application provides users with comprehensive access to the system's functionalities, allowing them to interact with the data and models effectively. Below are the key screens of the application, along with their functionalities:

Model Accuracy The "Model Accuracy" screen displays the current accuracy of each machine learning model used in the system. It provides details about the model name, the type of accuracy metric used (e.g., MAE, MSE), the specific metric being monitored (e.g., `http_server_requests`, `jvm_memory_committed`), the accuracy value, the training date, and actions to edit or delete the model. This screen allows users to download the accuracy data as a CSV file for further analysis. Figure 2 shows the "Model Accuracy" screen.

Model Metrics The "Model Metrics" screen lists all the metrics currently being monitored by the system. Each metric is displayed with its name and value type (e.g., Double). This screen provides an overview of the various performance indicators being tracked to assess the health and efficiency of the monitored applications. Figure 3 shows the "Model Metrics" screen.

Records The "Records" screen displays the latest prediction records made by the system. It shows the metric names and their predicted values. Users can add new prediction records using the "Add New Record" button. This screen allows users to quickly view and manage the most recent predictions, ensuring they can keep track of important metrics in real-time. Figure 4 shows the "Records" screen.

Model Name	Accuracy Name	Metric Name	Accuracy Value	Training Date	Action
GradientBoostingRegressor	MAE	http_server_requests	75.19%	2024-07-14	Edit Delete
SVR	MSE	jvm_memory_committed	94%	2024-07-14	Edit Delete
SVR	MAE	jvm_memory_committed	89%	2024-07-14	Edit Delete
GradientBoostingRegressor	MSE	jvm_threads_states	76%	2024-07-14	Edit Delete

Figure 2. Model Accuracy Screen

5.4 Results of the main metrics

This subsection provides a detailed analysis of the key performance metrics used to evaluate the models in this study. By focusing on the R² Score and Mean Squared Error (MSE), we can gain a comprehensive understanding of each model’s predictive accuracy and reliability.

5.4.1 R² Score

The results presented in Figure 5 provide a detailed analysis of the R² Score for the models GradientBoostingRegressor, RandomForestRegressor and SVR, applied to the metrics http_server_requests_active, jvm_memory_used, jvm_threads_started and system_cpu_usage. At Y it is the models that have been trained. There are 3 models for each metric, so they are repeated 4 times in the graph (as there are four metrics). The metrics are in the columns.

The distribution of R² Scores is heterogeneous, varying significantly with the model and metric considered. The GradientBoostingRegressor model shows superiority in the http_server_requests_active metric, with a score close to unity, indicating a remarkable ability to explain the variation in the data. This finding reinforces its effectiveness in capturing the dynamics intrinsic to the data, making it suitable for scenarios that demand high predictive accuracy.

The standard deviations of the R² Score values are illustrated in Figure 6, showing minimal variation and indicating consistent performance across different models and metrics. Specifically, the GradientBoostingRegressor exhibited the smallest standard deviation, emphasizing its reliability.

In contrast, the SVR model shows inferior performance, particularly in the jvm_threads_started metric, suggesting a suboptimal adaptation to this specific metric compared to the other models analyzed. These insights are fundamental for directing methodological selections in future research, promoting improvements in predictive models within the domain of system observability.

This trend analysis is vital for the evolution of monitoring and quality assurance practices in complex IT infrastructures,

offering a significant contribution to the field of system observability. The results corroborate the central hypothesis of this study, reaffirming the potential of applying machine learning techniques to improve observability and proactive systems management.

5.4.2 Mean Squared Error (MSE)

By evaluating the Mean Square Error (MSE) values illustrated in Figure 7, we can see a detailed overview of the models’ performance in relation to the metrics under study. The jvm_memory_used metric, in particular, shows a notable variation in MSE between the models, with RandomForestRegressor showing a considerably lower value when compared to SVR. This result indicates that RandomForestRegressor is more effective at capturing the essence of the data, offering more accurate predictions that are closer to reality, making it the most suitable selection for this specific metric. On the other hand, SVR has a high MSE for the same metric, pointing to lower prediction accuracy. This contrast emphasizes the importance of careful model selection, which must be congruent with the intrinsic characteristics of each type of data in order to achieve the best predictive effectiveness.

The standard deviations of the Mean Squared Error values are illustrated in Figure 8, where the RandomForestRegressor displayed lower variance compared to other models, particularly for the jvm_memory_used metric, highlighting its accuracy and robustness in prediction.

5.5 Summary of average results

This section provides a comprehensive overview of the performance metrics for both regression and classification models used in the study. By summarizing these metrics, it is possible to gain valuable insights into the predictive capabilities and effectiveness of each model, allowing for a thorough assessment of their strengths and weaknesses in different scenarios.

Metric Name	Value Type
disk.free	Double
hikaricp.connections.acquire	Double
http.server.requests	Double
http.server.requests.active	Double
jvm.buffer.count	Double
jvm.buffer.memory.used	Double
jvm.buffer.total.capacity	Double
jvm.classes.loaded	Double
jvm.compilation.time	Double
jvm.gc.memory.allocated	Double

Figure 3. Model Metrics Screen

Metric Name	Predict Value
http.server.requests	48
system.cpu.usage	57%

Figure 4. Records Screen

5.5.1 Regression Models - First Layer

The performance metrics of the regression models, including the Random Forest Regressor, Gradient Boosting Regressor and SVR, are summarized in Table 5. Each metric provides valuable insights into the predictive capabilities of the models, allowing for a comprehensive assessment of their effectiveness.

Starting with the MSE, the Gradient Boosting Regressor outperforms both the Random Forest Regressor and the SVR, achieving the lowest MSE of 6.98. Random Forest Regressor follows closely with an MSE of 8.26, while SVR displays the highest MSE of 22.76. In the case of MAE, similar to MSE, Gradient Boosting Regressor shows the lowest MAE of 3.19, indicating superior performance in minimizing prediction errors. The Random Forest Regressor also does well, with an MAE of 3.26, while the SVR shows a higher MAE of 9.98.

Moving on to the coefficient of determination (R^2), the Random Forest Regressor achieves the highest R^2 score of

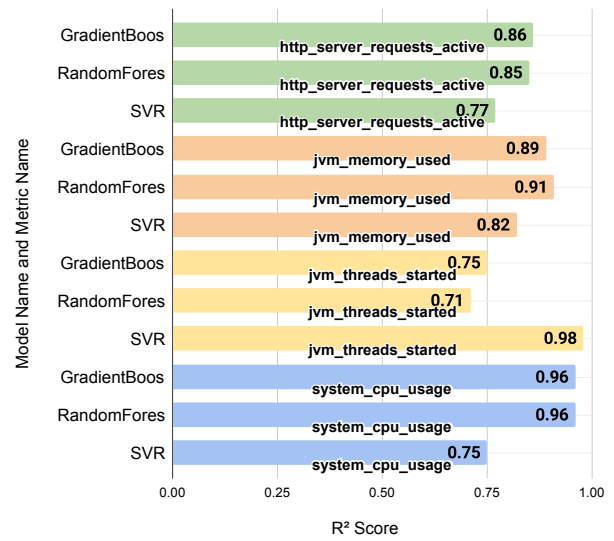


Figure 5. Accuracy of models according to R^2 Score

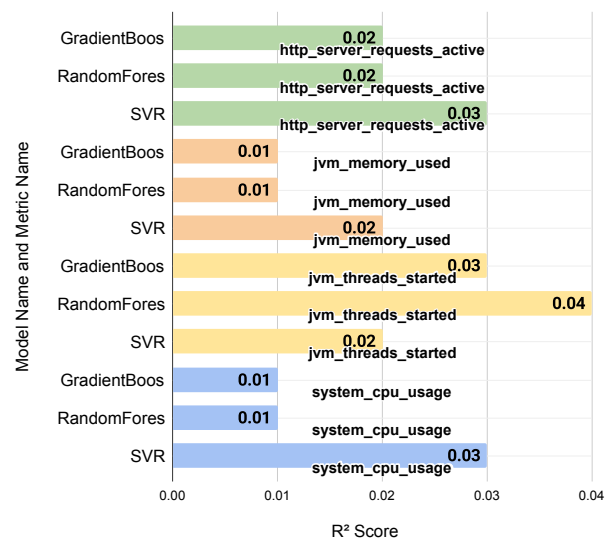


Figure 6. Standard Deviation in Accuracy of models according to R^2 Score

0.9, followed by the Gradient Boosting Regressor with 0.86. The SVR demonstrates a comparatively lower R^2 score of 0.68. Finally, in Explained Variance accuracy, the Random Forest Regressor exhibits the highest Explained Variance of 0.87, followed by the Gradient Boosting Regressor with 0.78. The SVR demonstrates the lowest Explained Variance of 0.61, indicating that it captures a smaller proportion of the variance in the data compared to the other models.

Analysis of performance metrics highlights the strengths and weaknesses of each regression model. Although Random Forest Regressor and Gradient Boosting Regressor generally outperform SVR on several metrics.

5.5.2 Classification Models - Second Layer

Table 6 shows the performance metrics of the classification models, including the Random Forest Classifier, Gradient Boosting Classifier and Logistic Regression. Each metric evaluates the ability of each model to correctly classify the data, taking into account the calculations presented above.

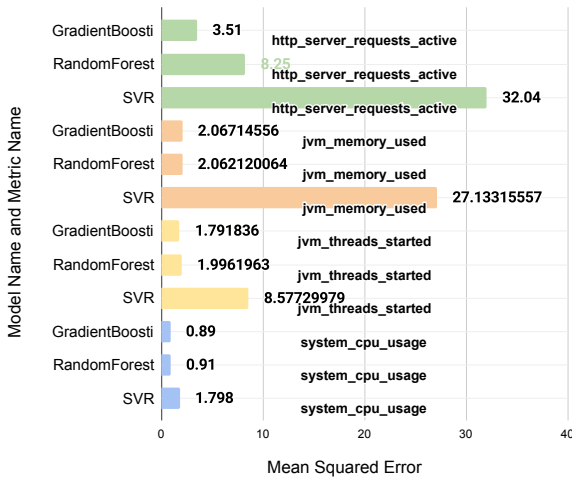


Figure 7. Model accuracy according to Mean Squared Error (MSE)

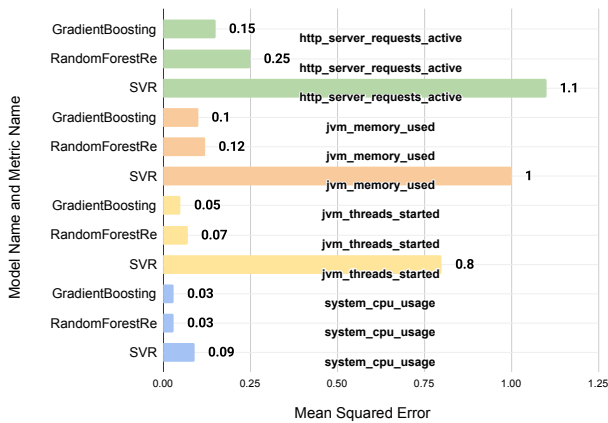


Figure 8. Standard Deviation in Model accuracy according to Mean Squared Error (MSE)

The MSE shows that the Gradient Boosting Classifier has the best metric, with a value of 1.02, followed by Logistic Regression with an MSE of 2.87. The Random Forest Classifier has an intermediate MSE of 2.9. The MAE results follow the same pattern, with the Gradient Boosting Classifier showing the lowest MAE of 0.002, followed by the Random Forest Classifier with an MAE of 0.82. Logistic Regression shows the highest MAE of 1.09.

As for the coefficient of determination (R^2), the Random Forest Classifier has the highest R^2 of 0.93, followed by Logistic Regression with 0.92 and Gradient Boosting Classifier with 0.89. The Explained Variance has similar results, with the Random Forest Classifier showing the highest Explained Variance of 0.95, followed by the Gradient Boosting Classifier with 0.93 and the Logistic Regression with 0.92.

In summary, the analysis of the performance metrics of the classification models highlights that the Gradient Boosting Classifier and the Random Forest Classifier generally outperform the Logistic Regression in several metrics.

5.6 Model results on different hyperparameters

Figure 9 shows the influence of hyperparameters on the R^2 metric in different regression models: RandomForestRe-

Metric	Random Forest Regressor	Gradient Boosting Regressor	SVR
MSE	8.26	6.98	22.76
MAE	3.26	3.19	9.98
R^2	0.9	0.86	0.68
Explained Variance	0.87	0.78	0.61

Table 5. Regressor Models Performance Metrics

Metric	Random Forest Classifier	Gradient Boosting Classifier	Logistic Regression
MSE	2.9	1.02	2.87
MAE	0.82	0.002	1.09
R^2	0.93	0.89	0.92
Explained Variance	0.95	0.93	0.92

Table 6. Classifier Models Performance Metrics

gressor, GradientBoostingRegressor and SVR. Lower values of hyperparameters ($n_estimators$: 50-100, max_depth : 1-2, $min_samples_split$: 1-3, $min_samples_leaf$: 1-2, $learning_rate$: 1-25) represent a limitation in the complexity of the model and may result in a tendency to underfitting. On the other hand, higher values ($n_estimators$: 500-1000, max_depth : 100-200, $min_samples_split$: 10-30, $min_samples_leaf$: 10-20, $learning_rate$: 1-1000) increase the risk of overfitting, characterized by high variance and poor generalization. This point of overfitting ends up generating biased models, which are only optimal for the training data.

The optimal hyperparameter ranges ($n_estimators$: 100-200, max_depth : 5-10, $min_samples_split$: 2-5, $min_samples_leaf$: 1-4, $learning_rate$: 1-100) show a balance between the model’s ability to capture patterns in the data without overparameterization. These configurations achieve the highest R^2 scores, corroborating the predictive robustness and generalization capacity of the models.



Figure 9. Comparison of hyperparameters in regression models

Figure 10 details the variation of the R^2 metric as a function of the hyperparameters for the RandomForestClassifier, GradientBoostingClassifier, and LogisticRegression classification models. For the lower magnitude hyperparameters ($n_estimators$: 100-200, max_depth : 5-10, $learning_rate$: 0.1-0.2, C : 0.1, 0.2, 0.3), there is an adequacy of the mod-

els that suggest a capacity for generalization, although possibly with room for improvement in terms of adjustment to the training data. In contrast, the higher magnitude hyperparameters ($n_estimators$: 400-600, max_depth : 30-40, $learning_rate$: 1, C : 20, 200, 2000) reflect a potential for overfitting, where the models may have a high variance and, consequently, a reduced capacity for generalization.

The ideal values ($n_estimators$: 200-300, max_depth : 15-20, $learning_rate$: 0.1-0.2, C : 10, 100, 1000) indicate an optimized configuration of the hyperparameters, providing balanced models capable of capturing the complexity of the data while maintaining good generalization. This is evidenced by the higher R^2 scores, implying greater classification accuracy.

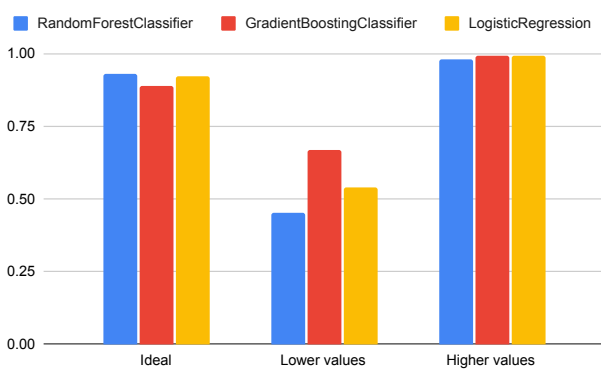


Figure 10. Comparison of hyperparameters in classification models

5.7 Scenarios for application

The proposed method excels in scenarios where early detection of performance degradation is crucial. For instance, in a high-frequency trading platform, predicting CPU usage spikes and memory leaks before they occur can prevent potential system crashes, ensuring uninterrupted trading operations. Similarly, in healthcare applications, predicting response times and system load can ensure that critical medical data processing is not delayed, which is vital for patient care. The ability to forecast anomalies and resource usage in advance allows for proactive resource allocation and incident management, significantly enhancing system reliability and performance.

In environments with dynamic workloads, such as e-commerce platforms during peak sales periods, the method's predictive capabilities enable automatic scaling of resources to meet increased demand, ensuring optimal performance and user experience. By integrating machine learning models with observability tools, this approach provides a comprehensive solution for proactive system management, making it invaluable for maintaining high availability and performance in complex, data-intensive environments.

6 Conclusions and Future Work

This study has consolidated the application of machine learning predictive models as a valuable tool for improving observability in complex IT systems. The microservices-based

architecture proved to be the right selection, with significant benefits in terms of scalability and maintenance. The GradientBoostingRegressor and RandomForestRegressor models proved to be particularly efficient, with the former achieving an R^2 Score of 0.86 when predicting HTTP request rates and the latter reducing the Mean Squared Error (MSE) by 2.06% for memory usage predictions when compared to traditional monitoring methods.

These advances highlight the models' ability to identify crucial patterns and anticipate anomalies with considerable accuracy, enabling more agile and informed interventions. However, challenges such as the need for fine-tuning models and improving training performance still persist. The complexity and computational cost of machine learning models demand special attention, indicating the need for ongoing research into optimization and efficiency.

Future work will explore strategies that can speed up the training process without compromising the accuracy of the models. This could include the application of more efficient algorithms, the use of specialized hardware, and data dimensionality reduction techniques. In addition, emphasis will be placed on implementing auto-tuning mechanisms that can simplify the selection of hyperparameters, making predictive models not only more agile but also accessible for wider adoption in IT production environments. Furthermore, modifying the application to be able to run more than one application on different servers is also mapped out future work.

These future guidelines aim to strengthen the proposition that integrating machine learning into observability is a technical enhancement that can take IT systems management to a new level of proactivity and resilience.

Declarations

Acknowledgements

A realização desta investigação foi parcialmente financiada por fundos nacionais através da FCT - Fundação para a Ciência e Tecnologia, I.P. no âmbito dos projetos UIDB/04466/2020 e UIDP/04466/2020

Authors' Contributions

These authors contributed equally to this work.

Competing interests

The authors declare that they have no competing interests.

Availability of data and materials

Data can be made available upon request.

References

Ahola, J. (2022). Cloud monitoring: cloud monitoring with dynatrace. Available at: <https://www.theseus.fi/handle/10024/786044> (Accessed: March 20, 2024).

- Aldi, F. et al. (2023). StandardScaler's potential in enhancing breast cancer accuracy using machine learning. *Journal of Applied Engineering and Technological Science (JAETS)*, 5(1):401–413. DOI: 10.37385/jaets.v5i1.3080.
- Barth, W. (2008). *Nagios: System and network monitoring*. Book.
- Behnel, S. et al. (2010). Cython: The best of both worlds. *Computing in Science and Engineering*, 13(2):31–39. DOI: 10.1109/MCSE.2010.118.
- Borré, A., Seman, L. O., Camponogara, E., Stefenon, S. F., Mariani, V. C., and Coelho, L. S. (2023). Machine fault detection using a hybrid CNN-LSTM attention-based model. *Sensors*, 23(9):4512. DOI: 10.3390/s23094512.
- Chakrabarty, N. et al. (2019). *Flight arrival delay prediction using gradient boosting classifier*. Springer Singapore. DOI: 10.1007/978-981-13-1498-8_57.
- Chakraborty, Mainak, K. A. P. (2021). *Grafana*, pages 187–240. Apress. DOI: 10.1007/978-1-4842-6888-9.
- Christodoulou, E. et al. (2019). A systematic review shows no performance benefit of machine learning over logistic regression for clinical prediction models. *Journal of clinical epidemiology*, 110:12–22. DOI: 10.1016/j.jclinepi.2019.02.004.
- Corso, M. P., Stefenon, S. F., Singh, G., Matsuo, M. V., Perez, F. L., and Leithardt, V. R. Q. (2023). Evaluation of visible contamination on power grid insulators using convolutional neural networks. *Electrical Engineering*, 105:3881–3894. DOI: 10.1007/s00202-023-01915-2.
- da Silva, E. C., Finardi, E. C., and Stefenon, S. F. (2024). Enhancing hydroelectric inflow prediction in the Brazilian power system: A comparative analysis of machine learning models and hyperparameter optimization for decision support. *Electric Power Systems Research*, 230:110275. DOI: 10.1016/j.epr.2024.110275.
- Da Silva, M. D. and Tavares, H. L. (2015). *Redis Essentials*. Book.
- De Souza, P. R. R., Matteussi, K. J., Veith, A. D. S., Zanchetta, B. F., Leithardt, V. R. Q., Murciego, □. L., De Freitas, E. P., Anjos, J. C. S. D., and Geyer, C. F. R. (2020). Boosting big data streaming applications in clouds with burstflow. *IEEE Access*, 8:219124–219136. DOI: 10.1109/ACCESS.2020.3042739.
- dos Santos, G. H., Seman, L. O., Bezerra, E. A., Leithardt, V. R. Q., Mendes, A. S., and Stefenon, S. F. (2021). Static attitude determination using convolutional neural networks. *Sensors*, 21(19):6419. DOI: 10.3390/s21196419.
- Dossot, D. (2014). *RabbitMQ essentials*. Book.
- Dynatrace (2024). Dynatrace documentation. Available at: <https://docs.dynatrace.com/docs> (Accessed: July 18, 2024).
- Elango, S. et al. (2022). Extreme gradient boosting regressor solution for defly in drilling of materials. *Advances in Materials Science and Engineering*. DOI: 10.1155/2022/8330144.
- Hao, N., He, F., Xie, C., Tian, C., and Yao, Y. (2022). Non-linear observability analysis of multi-robot cooperative localization. *Systems & Control Letters*, 168:105340. DOI: 10.1016/j.sysconle.2022.105340.
- Jouppi, N., Young, C., Patil, N., and Patterson, D. (2018). Motivation for and evaluation of the first tensor processing unit. *IEEE Micro*, 38(3):10–19. DOI: 10.1109/MM.2018.032271057.
- Junior, R. L. R., Malde, S., Cazzaniga, C., Kastriotou, M., Letiche, M., Frost, C., and Rech, P. (2022). High energy and thermal neutron sensitivity of google tensor processing units. *IEEE Transactions on Nuclear Science*, 69(3):567–575. DOI: 10.1109/TNS.2022.3142092.
- Klaar, A. C. R., Stefenon, S. F., Seman, L. O., Mariani, V. C., and Coelho, L. S. (2023). Optimized EWT-Seq2Seq-LSTM with attention mechanism to insulators fault prediction. *Sensors*, 23(6):3202. DOI: 10.3390/s23063202.
- Kramer, O. and Kramer, O. (2016). *Scikit-learn*, pages 45–53. DOI: 10.1007/978-3-319-33383-0.
- Leithardt, V., Santos, D., Silva, L., Viel, F., Zeferino, C., and Silva, J. (2020). A solution for dynamic management of user profiles in iot environments. *IEEE Latin America Transactions*, 18(07):1193–1199. DOI: 10.1109/TLA.2020.9099759.
- Liu, Y., Wang, Y., and Zhang, J. (2012). *New machine learning algorithm: Random forest*, volume 3. Springer Berlin Heidelberg. DOI: 10.1007/978-3-642-34062-8_32.
- McKinney, W. (2018). Python para análise de dados: Tratamento de dados com pandas, numpy e ipython. Book.
- Milani, A. (2008). *Postgresql-guia do programador*. Book.
- Min, L., Alnowibet, K. A., Alrasheedi, A. F., Moazzen, F., Awwad, E. M., and Mohamed, M. A. (2021). A stochastic machine learning based approach for observability enhancement of automated smart grids. *Sustainable Cities and Society*, 72:103071. DOI: 10.1016/j.scs.2021.103071.
- Moreno, S. R., Seman, L. O., Stefenon, S. F., dos Santos Coelho, L., and Mariani, V. C. (2024). Enhancing wind speed forecasting through synergy of machine learning, singular spectral analysis, and variational mode decomposition. *Energy*, 292:130493. DOI: 10.1016/j.energy.2024.130493.
- Noetzold, D. (2024a). Healthcheck analytics platform. Available at: https://github.com/DarlanNoetzold/healthcheck_API (Accessed: April 02, 2024).
- Noetzold, D. (2024b). Healthcheck training. Available at: <https://x.gd/UdzEV> (Accessed: April 06, 2024).
- Noetzold, D. et al. (2023). Use of spyware integrated with prediction models for computer monitoring. In *2023 18th Iberian Conference on Information Systems and Technologies (CISTI)*. IEEE. DOI: 10.23919/CISTI58278.2023.10211594.
- Olups, R. (2016). *Zabbix network monitoring*. Book.
- Ribeiro, M. H. D. M., da Silva, R. G., Moreno, S. R., Canton, C., Larcher, J. H. K., Stefenon, S. F., Mariani, V. C., and dos Santos Coelho, L. (2024). Variational mode decomposition and bagging extreme learning machine with multi-objective optimization for wind power forecasting. *Applied Intelligence*, 54:3119–3134. DOI: 10.1007/s10489-024-05331-2.
- Rodriguez-Galiano, V. et al. (2015). Machine learning predictive models for mineral prospectivity: An evaluation of neural networks, random forest, regression trees and support vector machines. *Ore Geology Reviews*, 71:804–818. DOI: 10.1016/j.oregeorev.2015.01.001.

- Singh, G., Stefenon, S. F., and Yow, K.-C. (2023). Interpretable visual transmission lines inspections using pseudo-prototypical part network. *Machine Vision and Applications*, 34(3):41. DOI: 10.1007/s00138-023-01390-6.
- Stefenon, S. F., Seman, L. O., Aquino, L. S., and dos Santos Coelho, L. (2023a). Wavelet-Seq2Seq-LSTM with attention for time series forecasting of level of dams in hydroelectric power plants. *Energy*, 274:127350. DOI: 10.1016/j.energy.2023.127350.
- Stefenon, S. F., Seman, L. O., da Silva, L. S. A., Mariani, V. C., and dos Santos Coelho, L. (2024). Hypertuned temporal fusion transformer for multi-horizon time series forecasting of dam level in hydroelectric power plants. *International Journal of Electrical Power & Energy Systems*, 157:109876. DOI: 10.1016/j.ijepes.2024.109876.
- Stefenon, S. F., Seman, L. O., Mariani, V. C., and Coelho, L. S. (2023b). Aggregating prophet and seasonal trend decomposition for time series forecasting of Italian electricity spot prices. *Energies*, 16(3):1371. DOI: 10.3390/en16031371.
- Stefenon, S. F., Seman, L. O., Sopelsa Neto, N. F., Meyer, L. H., Mariani, V. C., and Coelho, L. d. S. (2023c). Group method of data handling using Christiano-Fitzgerald random walk filter for insulator fault prediction. *Sensors*, 23(13):6118. DOI: 10.3390/s23136118.
- Stefenon, S. F., Singh, G., Souza, B. J., Freire, R. Z., and Yow, K.-C. (2023d). Optimized hybrid YOLOu-Quasi-ProtoPNet for insulators classification. *IET Generation, Transmission & Distribution*, 17(15):3501–3511. DOI: 10.1049/gtd2.12886.
- Surek, G. A. S., Seman, L. O., Stefenon, S. F., Mariani, V. C., and Coelho, L. S. (2023). Video-based human activity recognition using deep learning approaches. *Sensors*, 23(14):6384. DOI: 10.3390/s23146384.
- Tarek, Z. et al. (2023). Wind power prediction based on machine learning and deep learning models. *Computers, Materials and Continua*, 75(1). DOI: 10.32604/cmc.2023.032533.
- Turnbull, J. (2018). *Monitoring with Prometheus*. Book.
- Vos, G., Trinh, K., Sarnyai, Z., and Azghadi, M. R. (2023). Generalizable machine learning for stress monitoring from wearable devices: A systematic literature review. *International Journal of Medical Informatics*, 173:105026. DOI: 10.1016/j.ijmedinf.2023.105026.
- Webb, P. et al. (2013). Spring boot reference guide. Available at: <https://mackvord.github.io/PDF/spring-boot-reference.pdf> (Accessed: March 25, 2024).
- Yamasaki, M., Freire, R. Z., Seman, L. O., Stefenon, S. F., Mariani, V. C., and dos Santos Coelho, L. (2024). Optimized hybrid ensemble learning approaches applied to very short-term load forecasting. *International Journal of Electrical Power & Energy Systems*, 155:109579. DOI: 10.1016/j.ijepes.2023.109579.
- Zabbix (2024). Zabbix documentation. Available at: <https://www.zabbix.com/manuals> (Accessed: July 18, 2024).
- Zhang, F. and O'Donnell, L. J. (2020). *Support vector regression*, pages 123–140. Academic Press. DOI: 10.1016/B978-0-12-815739-8.00007-9.