Models and Technologies for Autoscaling Based on Machine Learning for Microservices Architecture

Serhiy Semerikov[1], Dmytro Zubov[2], Andrey Kupin[1], Maxim Kosei[1] and Vladyslav Holiver[1]

[1] *Kryvyi Rih National University, Vitaly Matusevich 11, Kryvyi Rih, 50027, Ukraine*
[2] *University of Central Asia, 125/1 Toktogul Street, Bishkek, 720001, Kyrgyzstan*

Abstract

The subject of the research in the article is machine learning processes in web service systems used for providing online services. The subject of the study is methods and tools for auto-scaling these web services using machine learning. The evolution of web services, their structure including development history, scaling options, key concepts of microservices architecture, and general principles of artificial intelligence and machine learning are analyzed, providing an important foundation for understanding technological innovations and potential enhancements for web services. The most significant aspects of applying machine learning in microservices architecture are identified, including various design patterns and machine learning models, which form the basis for improving the efficiency and capabilities of complex systems. Relevant mathematical models and necessary software are proposed.

Keywords

Microservices architecture, artificial intelligence, machine learning, deep learning, SAGA, CRUD, CQRS, API gateway, circuit breaker, Python, containers, Docker, Ubuntu.

1. Introduction

Web services are an essential part of the modern Internet. They enable web applications to interact with each other, regardless of what platforms or programming languages they are written in. This makes web services a valuable tool for developers who want to create flexible and scalable web applications [1, 2]. Web services are used in a wide range of applications, including e-commerce systems, banks, mobile operators, and web stores. For example, e-commerce systems often use web services to process payments, deliver goods, and provide customer support, banks use web services to provide financial services such as banking, loans, and deposits, mobile operators use web services to manage their networks and provide services to their customers, and web stores use web services to catalog products, process orders, and deliver goods. The significance of web services is growing every year, so it is becoming crucial to ensure the scalability of web services, i.e., the ability of the system to increase the processing amount with the increase in the number of users [3].

The application of artificial intelligence (AI) to automate the scaling of web services is an active area of research. Scientists and engineers are developing new AI methods and algorithms that can help improve the efficiency and accuracy of automated scaling in Microservices architecture (MSA). Here are a few specific examples of how AI can be used to automate web service scaling: Amazon Web Services (AWS) uses AI for its Auto Scaling system, which automatically deploys or shuts down servers based on load; Google Cloud Platform (GCP) uses AI for its Cloud Autoscaling system, which has similar functionality to AWS' Auto Scaling; Microsoft Azure uses AI for its Azure Autoscale system, which also has similar functionality.

COLINS-2024: 8th International Conference on Computational Linguistics and Intelligent Systems, April 12–13, 2024, Lviv, Ukraine

semerikov@gmail.com (S. Semerikov); dzubov@ieee.org (D. Zubov); kupin@knu.edu.ua (A. Kupin); kosei@knu.edu.ua (M. Kosei); holivervlad@gmail.com (V. Holiver)

0000-0003-0789-0272 (S. Semerikov); 0000-0002-5601-7827 (D. Zubov); 0000-0001-7569-1721 (A. Kupin); 0009-0008-3445-5675 (M. Kosei); 0000-0002-8276-5992 (V. Holiver)

Autoscaling based on machine learning (ML) can be utilized not only for web services but also for other types of applications such as mobile apps, embedded systems, and more. For instance, autoscaling can be employed for mobile applications that experience a high volume of server requests, such as online shopping apps or social networks. ML-based autoscaling of web services allows for automatically adjusting server resources based on the workload of the web service. This is achieved using ML algorithms that analyze service monitoring data and forecast future workload. For example, classification algorithms, regression, clustering, or neural networks can be employed [4-7].

Advantages of using ML-based auto-scaling methods and tools:

- high accessibility and reliability of the system due to horizontal scaling and the use of microservice architecture;
- efficient use of resources due to automatic scaling of resources based on the actual load;
- flexibility and scalability of the system due to the ability to add new servers and containers as needed and scale individual services depending on the load;
- high performance and avoidance of unpredictable failures through the use of machine learning to predict future load and automatically scale server resources based on the current load.

Although machine learning and container-based autoscaling methods and tools have many advantages, they also have some disadvantages [8]. One of the main drawbacks is the complexity of implementing and customizing these tools. In addition, the use of machine learning can require significant computing resources, which can lead to increased hardware and system maintenance costs. Finally, autoscaling can be difficult when using third-party services such as databases or other tools that may not be compatible with autoscaling.

Moreover, it is worth noting that when employing autoscaling methods and tools for web services, security considerations must be taken into account. For example [9], when using containers, ensuring container security is essential as they may contain sensitive data. Additionally, network security on which the web service operates should be ensured as inadequate security measures can lead to system breaches and loss of confidential information.

Additionally, it is essential to consider that security is an ongoing process that requires continuous monitoring and updating of protection methods and tools. Therefore, when selecting methods and tools for autoscaling web services, security considerations must be taken into account, and continuous monitoring and system updates should be ensured to maintain the highest level of security for the web service.

When choosing autoscaling methods and tools, it is also essential to consider economic feasibility. The utilization of complex methods and tools may result in a significant increase in deployment and maintenance costs. Therefore, when selecting autoscaling methods and tools, it is necessary to consider their effectiveness and cost, as well as which methods and tools are most optimal for the specific system.

The aim of this research is to develop efficient methods and algorithms for autoscaling web services based on machine learning to ensure stable operation of services under varying workloads. The tasks include analyzing existing autoscaling methods, developing new algorithms, implementing them, and testing them on real web services.

2. Analyzing Patterns for Designing in Microservices Architecture (MSA)

MSA offers numerous advantages in terms of flexibility, scalability, and service independence, but it also introduces complexities associated with managing these distributed systems. Utilizing design patterns in MSA becomes exceedingly important as they offer ready-made and proven solutions to such challenges. Patterns help find optimal ways to manage data consistency, implement service-to-service relationships, monitoring, and scaling. They contribute to the creation of stable, efficient microservices applications, allowing developers to focus on application functionality rather than solving complex technical tasks.

In MSA, where system components are divided into separate services, situations may arise where one service successfully makes changes to the database while another service, depending on these changes, has not yet updated its data. This can lead to inconsistent data state in the system. Additionally, problems may arise during the execution of distributed transactions (Figure 1), where part of the transaction is executed successfully, while another part is not. This may be caused by network issues, software errors, or other unforeseen circumstances.

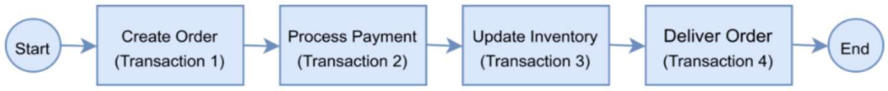

Figure 1: Example of a distributed transaction where transaction boundaries are crossed by multiple services and databases

To address issues with distributed transactions, the SAGA (Segregated Access of Global Atomicity) design pattern is used. This pattern is based on the idea of breaking down a large transaction into smaller sub-transactions, which are executed separately in each microservice (Figure 2). If one of the sub-transactions fails, a compensating transaction is applied to undo or adjust the changes made by the preceding transactions.

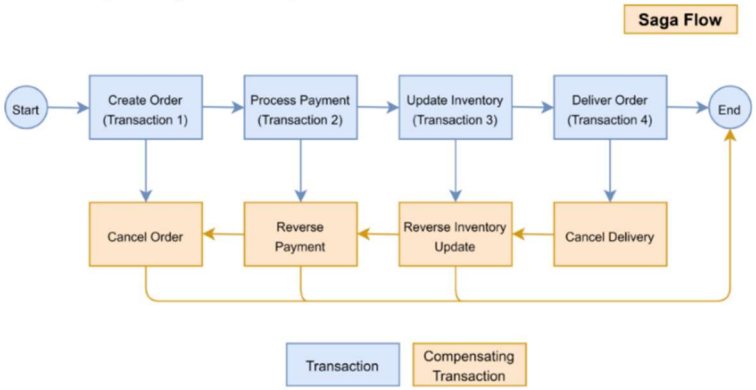

Figure 2: An example of a distributed transaction using the SAGA Flow pattern

The SAGA Execution Coordinator (SEC) is the central component for executing a SAGA flow (Figure 3). It contains the SAGA Log, which records the sequence of events of a distributed transaction. There are two types of SAGA pattern implementation: Choreography and Orchestration. In the SAGA Choreography pattern, each microservice involved in a transaction publishes an event that is processed by the next microservice. This is an interaction of services where each service receives events from others and responds to them, controlling the course of the transaction. To use this pattern, you need to decide whether the microservice will be part of SAGA. Accordingly, the microservice must use an appropriate framework to implement SAGA. In this pattern, the SAGA runtime coordinator can be embedded in the microservice or act as a separate component.

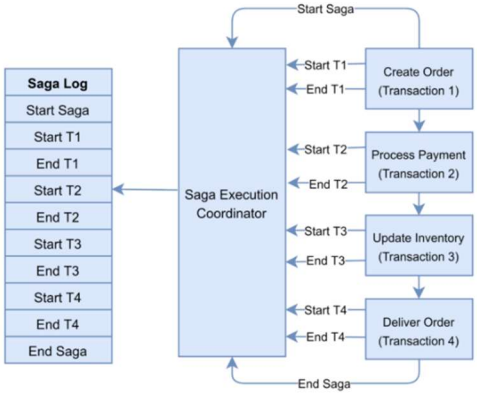

Figure 3: Saga Execution Coordinator Component

In SAGA Choreography, the flow is considered successful if all microservices successfully complete their local transactions and none of them report a failure (Figure 4). In case of failure, the microservice notifies the SAGA Execution Coordinator (SEC), through which corresponding compensating transactions are invoked. If the invocation of the compensating transaction fails, it is retried until successfully executed, with the compensating transaction being idempotent and capable of being retried.

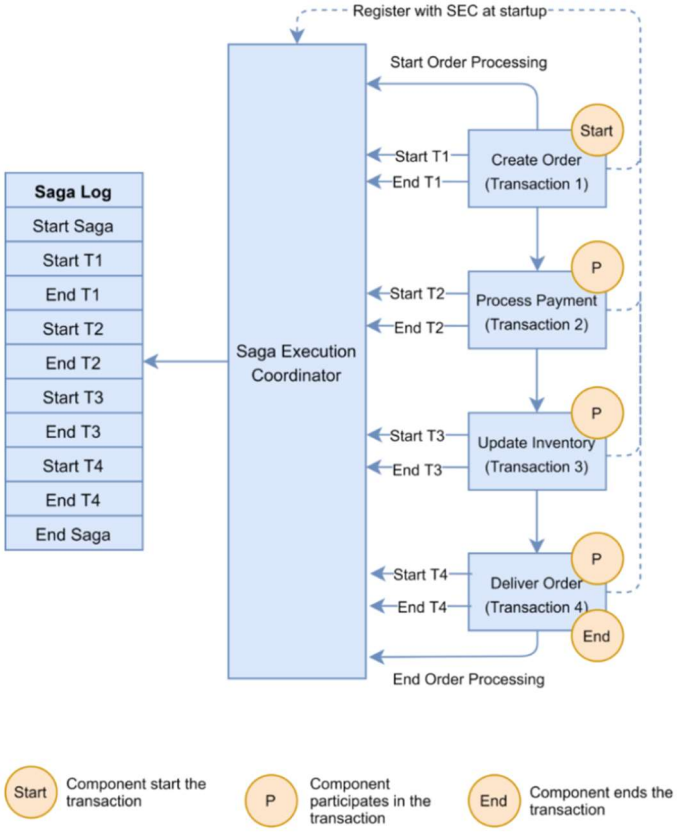

Figure 4: SAGA Choreography diagram illustrates the successful execution of a transaction

In traditional systems, especially in monolithic applications, it is very common to use a shared relational database hosted on the server side and accessible from the user application. This centralized database is accessed using Create-Read-Update-Delete (CRUD, Figure 5) operations. However, in today's complex MSA applications, especially when scaling the application, this traditional implementation creates a problem. When processing multiple CRUD requests to a database, table connections are created, which leads to a high probability of database locks. Blocking tables causes delays and competition for resources, which significantly affects the overall performance of systems.

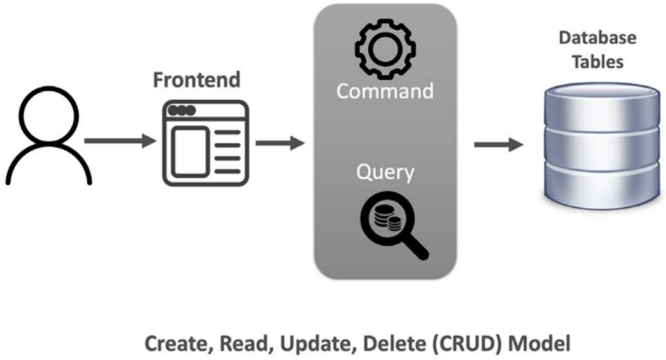

Figure 5: CRUD Pattern

Complex queries contain a large number of table connections and can block tables, preventing any write or update operations until the query is complete and the database unblocks the tables. Database read operations are usually performed several times more frequently than write operations, and in systems with intensive transaction execution, this problem can be exacerbated.

The CQRS (Command Query Responsibility Segregation) pattern divides one object into two objects. So, instead of executing both commands and queries for one object, we split this object into two objects - one for the command and one for the query. A command is an operation that changes the state of an object, and a query does not change the state of the system - it just returns the result. The object here is the Database, and this section of the database can be physical or logical.

However, the best practice is to have two physical databases for CQRS, but still, it is possible to use one physical database for both commands and queries. For example, the database can be divided into two logical representations - one for commands and the other for queries. When using two physical databases in CQRS, a replica is created from the primary database. The replica will need to be synchronized with the primary database for data consistency. Synchronization can be achieved by implementing Event-driven architecture (EDA), where a message broker handles all system events. The replica subscribes to the message broker, and whenever the primary database publishes an event in the message broker, the replica database synchronizes that specific change.

There will be a delay between the exact time the master database was actually changed and the time that change is reflected in the replica; the two databases will not be 100% consistent during this period, but they will be consistent over time after the change is synchronized. In CQRS, this synchronization is called final consistency synchronization. When applying CQRS in MSA, the latency of database operations is significantly reduced, and therefore, the performance of communication between individual services is significantly improved, which leads to an overall increase in system performance.

The type of database used for CQRS may vary depending on the business requirements for a particular service in the MSA. It may well be a relational database (RDB), a document-oriented database, a graph database, or another type of NoSQL database.

Microservices can communicate directly with each other without the need for a centralized manager (Figure 6). However, as the MSA system evolves and becomes more mature, and the number of microservices gradually increases, direct communication between microservices can lead to significant overheads, especially for calls that require multiple round trips between the API consumer and the API provider.

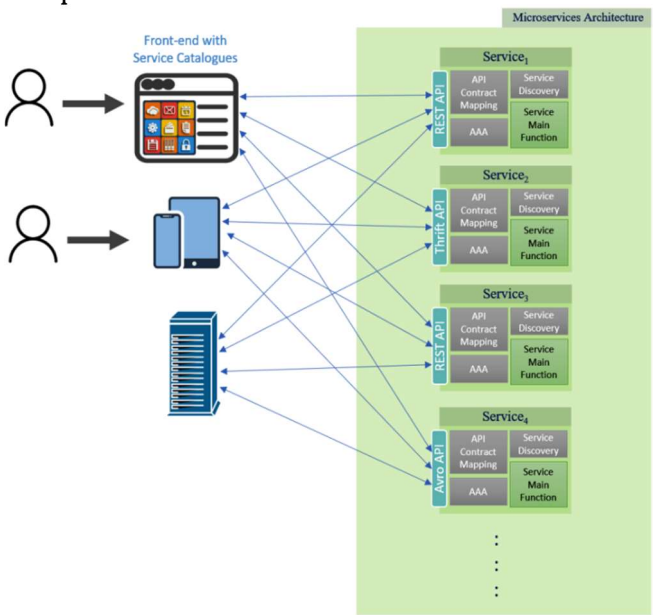

Figure 6: MSA System with Direct Service Interaction

Following the principle of microservices autonomy, each microservice can use its own technological stack and may communicate using a different API contract than other microservices in the same MSA system. For example, one microservice may only understand RESTful APIs with JSON data structure, while others may communicate only through Thrift APIs or Avro APIs.

Furthermore, the location (IP address and port for listening) of active microservice instances can dynamically change within the system based on microservice needs. Therefore, the system needs a mechanism to identify the destination to which the API consumer can send its calls. All of this requires additional built-in code in each microservice to help understand outdated APIs, discover the network location (Service Discovery) of other microservices in the system, and comprehend the overall needs of each microservice. However, such an approach violates the principle of microservices autonomy and leads to system complexity as with each new microservice added to the system, the number of potential interactions between microservices increases.

A more efficient approach to addressing these issues is to use the API Gateway pattern (Figure 7). Here, all microservices communicate with each other, receive API calls from consumers, and then transform the received data into a data structure and protocol that API providers can understand and process. Using the API Gateway significantly reduces direct one-to-one communication between services. Additionally, microservices are relieved from the code required to perform tasks such as API contract mapping, service discovery, and tasks related to controlling and providing access to system resources (AAA - Authentication-Authorization-Accounting).

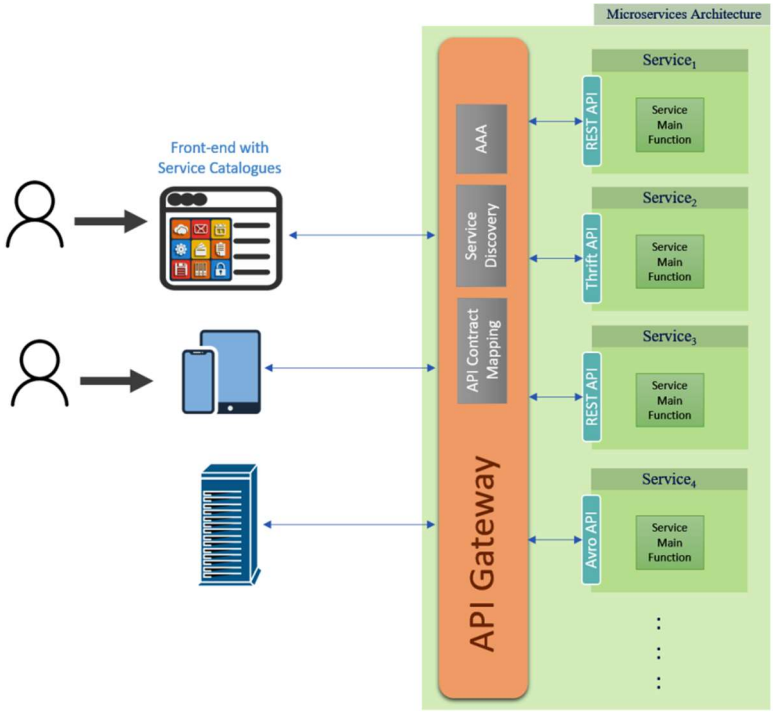

Figure 7: MSA System with API Gateway

Another important issue in MSA systems is the stability and reliability of executing business processes. Patterns like SAGA (Figures 2-3) are used to ensure that all transactions in a specific business process either all succeed or none of them do. However, this is insufficient for the reliable operation of an MSA system.

Let's consider a scenario (Figure 8) where the called microservice is too slow to react to API calls. Requests are successfully executed, but responses from the microservice end in a timeout. The user of the microservice may assume a failure of execution, and accordingly repeat the operation, which can be very problematic. In MSA systems, when creating a microservice, limited resources and threads are separated to avoid one particular microservice from taking up all the system resources.

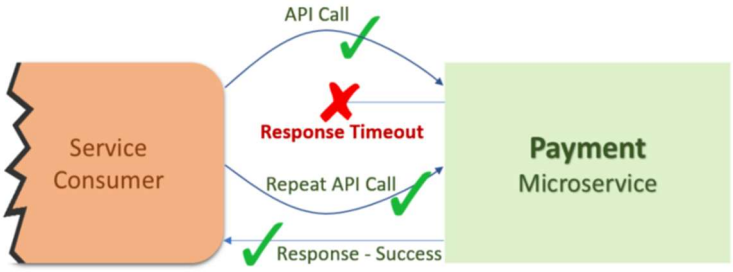

Figure 8: Example of interaction between microservices with excessively slow response time

Let's consider another scenario (Figure 9), where the Inventory microservice is part of a workflow and fails to respond to API calls for any reason. In this case, both the Order and Payment microservices will continue to wait for confirmation from the Inventory microservice before releasing their resources.

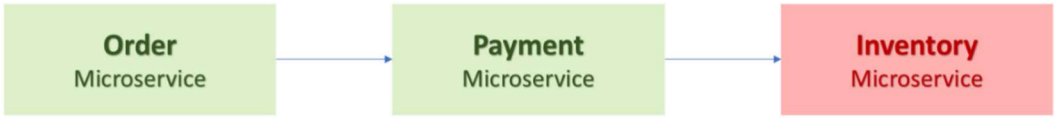

Figure 9: Example of interaction between microservices when one of the microservices does not respond to calls

Such scenarios in MSA systems can trigger a domino effect, leading to a cascading failure of multiple microservices, which in turn can cause a system-wide failure. To prevent a cascading failure of the system, the Circuit Breaker pattern is used.

The Circuit Breaker monitors the performance of the microservice using real traffic metrics. It analyzes parameters such as response time and the percentage of successful responses and determines the state of the microservice in real-time. If the microservice stops responding to calls, the Circuit Breaker transitions to an open state and immediately sends an error to the consumers of the microservice. When the Circuit Breaker detects that the microservice is experiencing issues, it still monitors and evaluates its state, but it transitions to a half-open state, allowing only a limited number of requests to pass through to the controlled microservice. If the breaker then detects normal behavior from the microservice, it transitions back to a closed state, and all requests are again routed for processing to the microservice.

3. Model Selection for Machine Learning (ML)

Among the most significant types of ML models are regression, neural network, and multiclass classification models [10]. Let's analyze their main features.

Regression models are modeling methods used to determine the relationship between independent and dependent variables. Typically, the outcome of a regression model is a continuous value, also known as a quantitative variable. Typical examples include predicting house prices based on their characteristics or forecasting sales of a particular product in a new store based on information about previous sales.

Before creating a regression model, it is necessary to understand the data and its structure. Most regression models use supervised learning (SL) methods. For regression models, the training data usually consists of a set of features and the values of the dependent variable, known as the label. Features are typically denoted as X, and labels as Y. Usually, the training data is divided into two subsets: training and testing sets. The training set typically consists of 70-80% of the total data, while the testing set contains the rest. This allows the model to learn from the training set and evaluate its performance on the testing set to assess its accuracy and quality. From the obtained results, we can draw conclusions about how our model performs on the dataset.

For a linear regression model to work effectively, our data must have a linear structure. The model uses this formula to train and learn from the data

$$y_i = \beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n, \quad (1)$$

where y_i is the final outcome (actual value) of the target parameter; $x_1,\ldots,x_n$ are input parameters; $\beta_0,\beta_1,\ldots,\beta_n$ are free parameters; n is number of data points.

In the case of linear regression, there are two common metrics used to evaluate the model. These are typically the Root Mean Square Error (RMSE) and the coefficient of determination R^2.

RMSE represents the standard deviation of the residual errors in predictions. Residual indicates the distance between actual data points and the regression line. The greater the average deviation of all points from the line, the higher the error. This indicates a weak model as it fails to capture the correlation between data points. This metric can be calculated using the following formula

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{N}}, \quad (2)$$

where $\hat{y}_i$ is the predicted value.

The coefficient of determination (R^2), measures the proportion of the variance in the dependent variable (Y), that can be explained by the independent variables (X). Essentially, it indicates how well the data fit the model. Unlike RMSE, which can be any number, R^2 is expressed as a number ranging from 0 to 1, making it easier to understand. The closer R^2 to 1, the better the correlation of the data. Although this is a useful indicator, values close to 1 percent are not always indicative of a strong model. The quality of the value depends on the specific application and the user's understanding of the data.

R^2 can be calculated using the formula

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

where $\bar{y}_i$ is the mean value for the entire dataset.

Many other metrics can assess the effectiveness of a regression model, but these two are sufficient to get an idea of how it performs. When creating and evaluating a model, it is important to visualize the data and the model, as this can reveal key insights. Graphs can help determine whether the model is overfitting (Figure 10) or underfitting (Figure 11).

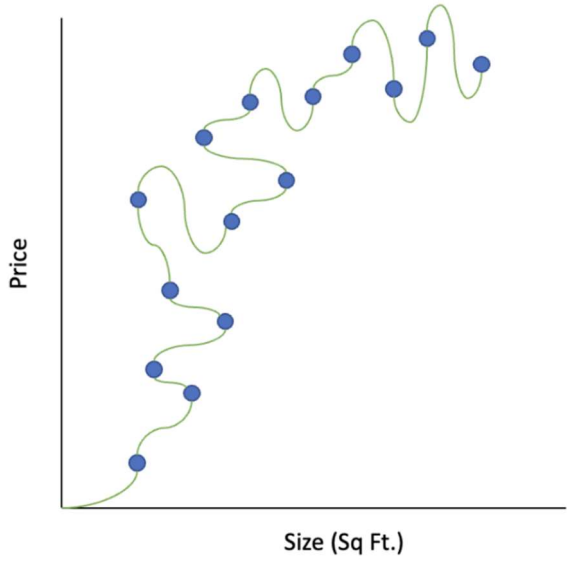

Figure 10: Example of an overfitted regression model

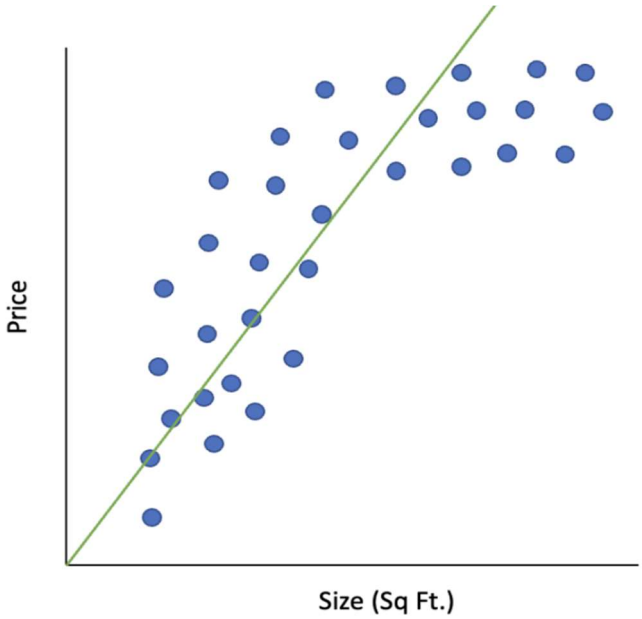

Figure 11: Example of an underfitting regression model

Multiclass classification is an ML task that involves classifying objects into multiple classes. Unlike binary classification, where an object can belong to only one of two classes, in multiclass classification, an object can belong to multiple classes. Multiclass classification models are considered versatile as they can be applied in both Supervised Learning (SL) and Unsupervised Learning (UL), whereas regression models are primarily used in SL. There are several regression models (such as logistic regression and support vector machines) that are also considered classification models because they use a threshold to partition the output of continuous values into different categories.

UL is a widespread application that is widely used. Although SL typically performs better and provides significant results since we know the expected output, most of the data we collect is unlabeled. It takes a lot of time and money for experts to review and label data. UL helps reduce costs and time by allowing models to attempt to determine labels for data and extract meaningful information from them. Sometimes, they can even outperform humans.

The number of categories at the output of the classification model determines its type (Figure 12). If the model has only two outputs (for example, dividing email messages into "spam" and "not spam"), then it is a binary classifier. In the case where the model has more than two outputs, it is considered a multiclass classifier.

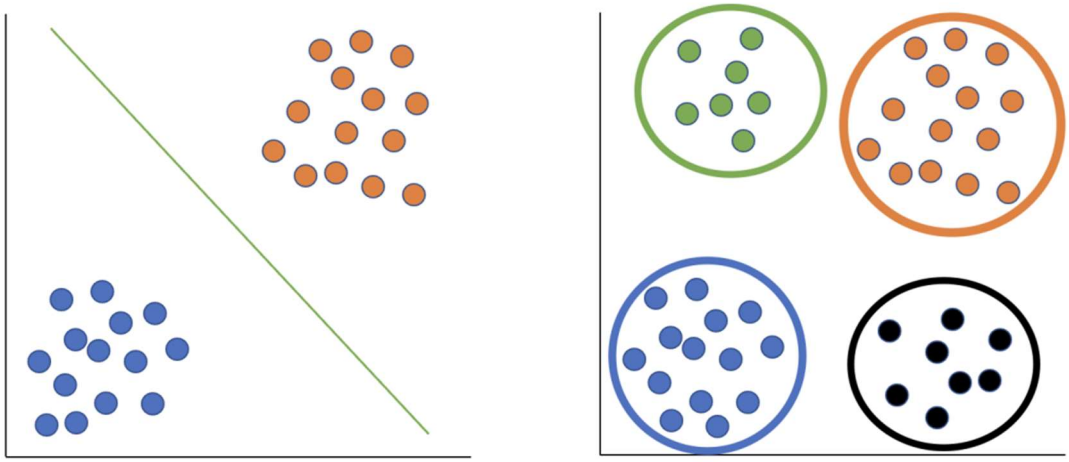

Figure 12: Binary and Multiclass Classifiers

Classifiers can be classified by the type of learning algorithm they use. There are two types of learning algorithms: lazy learning and eager learning. Lazy learning algorithms actually store the training data and wait until they receive new test data. Once they receive test data, the model classifies the new data based on the existing data. These types of algorithms require less time during training since new data can be continuously added without retraining the entire model, but they spend more time during classification as they need to go through all data points.

Main algorithms of this type include:

1) K-Nearest Neighbor algorithm (KNN).
2) Support Vector Machines algorithm (SVM).
3) Naive Bayes Classifier algorithm.

Eager learning algorithms, on the other hand, work oppositely. Each time new data is added to the model, it needs to be retrained. Although this takes more time compared to lazy learning algorithms, querying the model is much faster since they don't need to go through all data points.

Main algorithms of this type include:

1) Decision tree.
2) Artificial Neural Networks (ANN).

Neural networks can also be used for modeling and forecasting time series. The main advantage of neural networks compared to traditional methods lies in their ability to automatically detect complex dependencies in data and adapt to changes in the time series.

TLRN (Time-Lagged Recurrent Network) networks and the NNARX model (Nonlinear Nonparametric AutoRegressive model with eXogenous inputs) (Figure 13) are a subtype of Recurrent Neural Networks (RNNs), which are typically used for modeling time series, particularly for forecasting the next element of a time series based on previous elements. TLRN allows for considering time delays in the input data and preserving the network's previous states for better forecasting of future values.

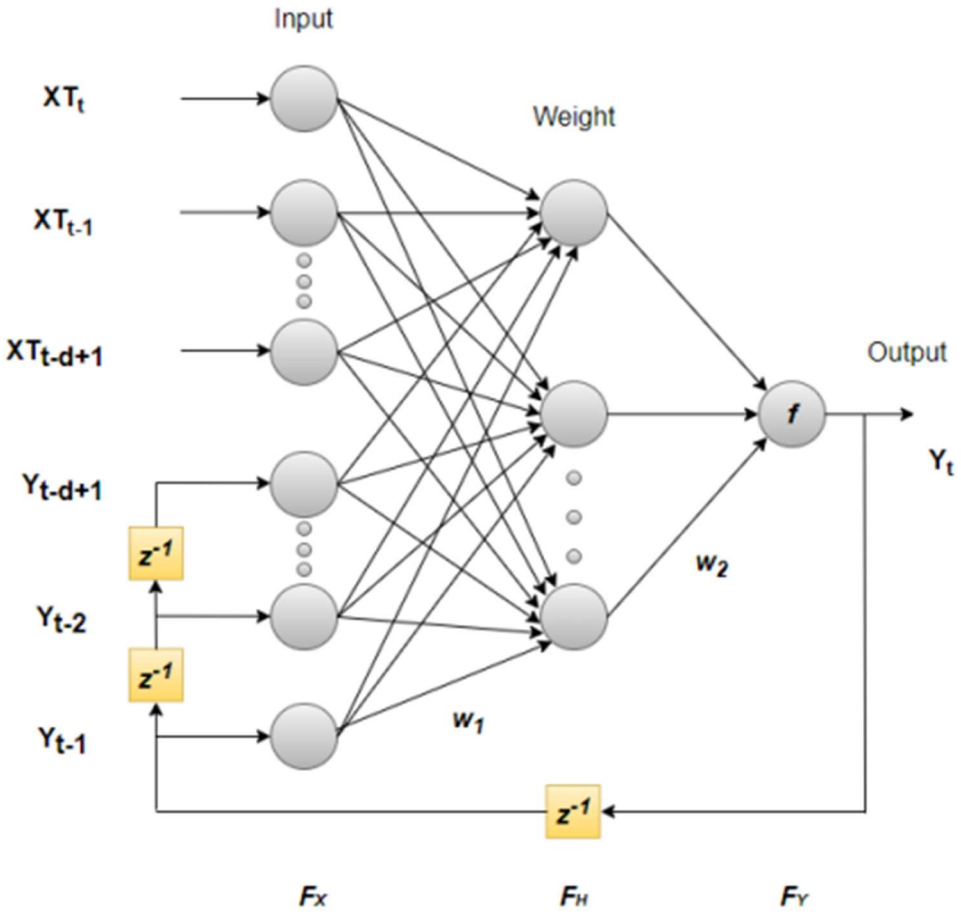

Figure 13: Architecture of NARX type recurrent network model

Unlike the feedforward artificial neural network, where data are processed in one direction from the input to the output layer without feedback, the RNN preserves information about previous data states using feedback loops, allowing the model to retain previous data states and use them for further data processing.

Also prevalent are recurrent neural networks such as LSTM (Long Short-Term Memory) and GRU (Gated Recurrent Unit). The architecture of LSTM was specifically designed to address the vanishing gradient problem that can occur during the backpropagation algorithm in traditional recurrent neural networks. For example, the LSTM Cell type architecture includes three gates: input, output, and forget gates (Figure 14).

The gating blocks can learn to open or close based on input data and the previous memory state, allowing the network to selectively retain or discard information over time.

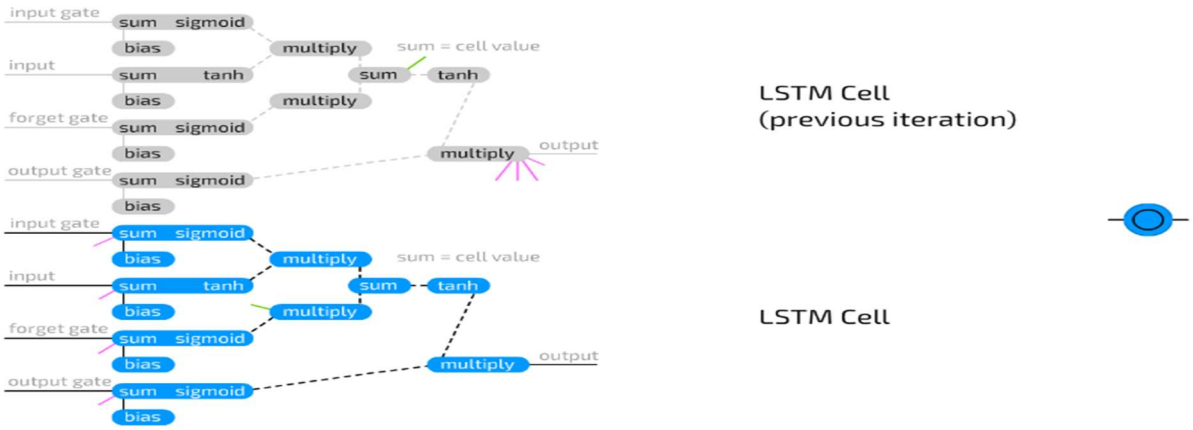

Figure 14: Architecture of LTSM Cell Network

4. Libraries for Machine Learning (ML) Models in Python

There are many programming languages used for creating ML models: MATLAB, R, and Python. Among them, Python has become the most popular programming language in the field of machine learning due to its versatility and the abundance of libraries [11] that facilitate the creation of machine learning models.

NumPy is a key library in developing machine learning models in Python because there is a lot of work with large multidimensional arrays involved in model creation. Since the bulk of the work involves transformations, splitting, and performing complex mathematical operations on these arrays, NumPy provides fast and efficient tools for this.

Matplotlib is an important library for visualizing results and model data. It provides an easy way to create plots, ranging from simple line plots to more complex ones such as contour plots and 3D plots. The popularity of this library is due to its ease of interaction with NumPy.

The Pandas library has become popular in the Python community due to its convenience and versatility in working with data stored in CSV files, which has been a recent trend. This library is used for data analysis, storing data in a tabular format. Users are provided with simple functions for preprocessing and manipulating data, allowing them to adapt the data to their needs. Additionally, Pandas is useful for working with time series data, which is an important aspect when building forecasting models. The TensorFlow and Keras libraries are the foundation for building deep learning models. Although both can be used independently, Keras is utilized as an interface for the TensorFlow framework, allowing users to easily create powerful deep learning models.

TensorFlow, developed by Google, serves as the backend for building machine learning models. It operates by creating static data flow graphs that specify how data moves through deep learning. The graph consists of nodes and edges, where nodes represent mathematical operations. It passes this data using multidimensional arrays known as tensors.

Keras, which was later integrated with TensorFlow, can be considered as a frontend for designing deep learning models. It was implemented with user convenience in mind, allowing them to focus on designing their neural network models without delving into the complex details of the backend. It resembles object-oriented programming as it replicates the style of object creation. Users can freely add different types of layers, activation functions, and more. They can even utilize pre-built neural networks for easy training and testing.

PyTorch is another machine learning framework created by Meta, formerly known as Facebook. Similar to Keras/TensorFlow, it allows users to create machine learning models. This framework is well-suited for natural language processing (NLP) and computer vision tasks, but it can be configured for most applications. What makes PyTorch unique is its dynamic computational graph. It has a module called Autograd, which enables automatic differentiation dynamically, unlike TensorFlow, where it is static. Additionally, PyTorch is more aligned with the Python programming language, making it easier to understand and utilize Python's useful features such as parallel programming.

SciPy is a library designed for scientific computing. It contains many built-in functions and methods for linear algebra, optimization, and integration, which are often used in machine learning. This library is useful for calculating certain statistical indicators and transformations when building machine learning models.

Scikit-Learn is a machine learning library that is an extension of SciPy and built using NumPy and Matplotlib. It includes many built-in machine learning models such as Random Forest, k-means clustering, and Support Vector Machine (SVM).

5. Integration of machine learning (ML) into MSA

Having analyzed the core concepts and approaches used in MSA and ML, the next step is to synthesize approaches for integrating ML into the MSA system. The space for integrating machine learning into MSA systems is quite extensive and can be utilized for many different scenarios, but four main integration areas can be highlighted [12-14].

The first area is forecasting system load, which allows determining when microservices experience higher-than-usual loads and taking measures to prevent system failures (Figure 15). Forecasting system load is a common issue when working with web services. MSA has an advantage over monolithic systems because resources are allocated separately for each microservice, simplifying maintenance and scalability. However, there are situations in MSA where a microservice experiences a high load, leading to a cascading effect where failures spread to other microservices.

With the help of ML, a model can be trained using various features, such as critical microservices response time, and used to detect patterns in the operation of the MSA system. Similar to the Circuit Breaker, this model can promptly determine whether a microservice is under heavy load and address this issue before it becomes critical and starts negatively impacting other microservices.

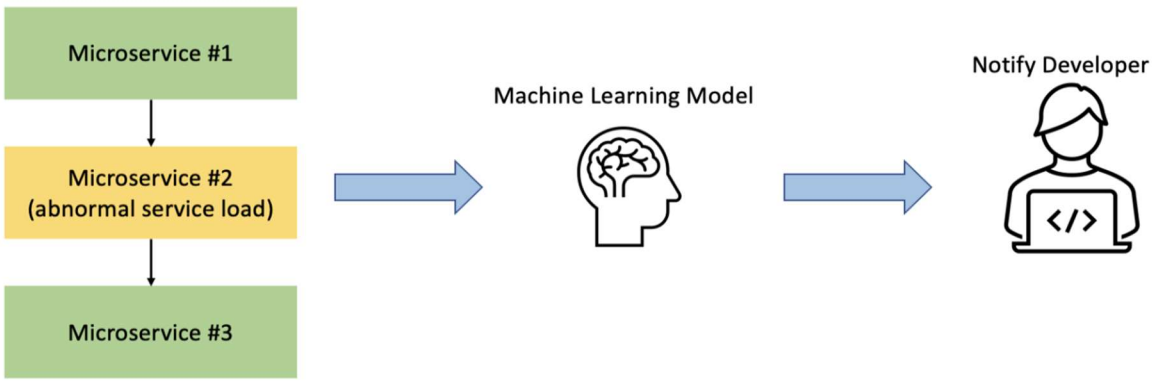

Figure 15: System Load Forecasting Model

The second area is forecasting system performance degradation. This is similar to forecasting system load but has a more specific goal - to identify issues that may lead to a decrease in system performance or reliability (Figure 16).

Like the system load forecasting model, we can build a model to detect anomalies in MSA that may lead to system performance degradation. Instead of focusing solely on the load for a specific microservice, it is possible to study the entire MSA and identify various patterns in how it operates overall.

MSA systems may experience varying loads and errors during specific times and periods. For instance, an MSA system may encounter spikes in requests during certain periods, such as holidays and seasonal events, when the number of users can sharply increase. By allowing the model to learn and understand the MSA system and how it operates over time, we can prepare the model to better detect anomalies and prevent false positives.

Moreover, rather than tracking individual microservices, it is necessary to evaluate clusters of microservices and how they interact with the entire MSA system. This way, there is an opportunity to identify specific bottlenecks and errors that may arise at the scale of the entire system.

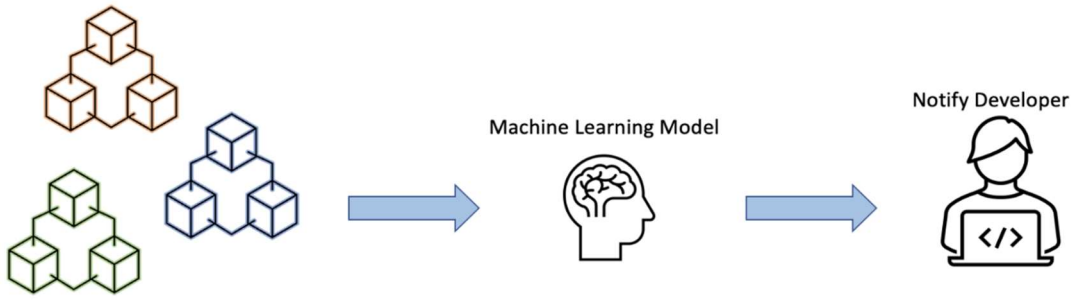

Figure 16: System Productivity Degradation Prediction Model

The third area of ML application is system security: In the era of cybersecurity, it is important to have the ability to protect your MSA system from targeted attacks. By studying the behavior of your MSA system, the model can predict and detect attacks that may threaten the security of the system.

Machine learning is successfully used in the field of cybersecurity. With the development of more sophisticated hacker attacks, the issue of protecting the MSA system becomes relevant.

Machine learning simplifies the creation of reliable models that can analyze and predict attacks before they can impact the operation of the system.

For example, Denial of Service (DoS) attacks, aimed at disrupting users' access to certain services, are becoming increasingly sophisticated with the advancement of technology. With machine learning, it is possible to recognize DoS attacks within the context of the MSA system. Thus, it can determine whether our system is susceptible to such MSA attacks and alert the security team or implement countermeasures to combat specific attacks and maintain system integrity.

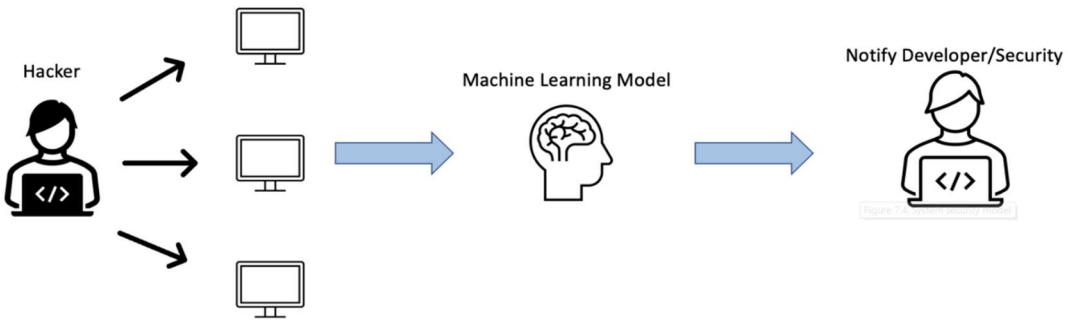

Figure 17: System Protection Model

The fourth area of ML application is system resource planning. As the system grows and evolves, it is important to properly allocate resources and adapt to the system's needs. With machine learning, it is possible to learn which services require more resources and how much resources are needed for effective system scalability (Figure 18). Part of the system self-healing process involves allocating resources to certain microservices in case of system growth and expansion in the MSA system.

Over time, as the number of users increases, resulting in an increase in the number of requests and load on the system, incorrect problem identification and erroneous resource planning and allocation may occur. However, the application of a forward-looking ML model, which can track the gradual growth of the MSA system and identify when certain services require additional resources, will help avoid such mistakes and significantly improve system reliability, as it can accurately and efficiently allocate resources.

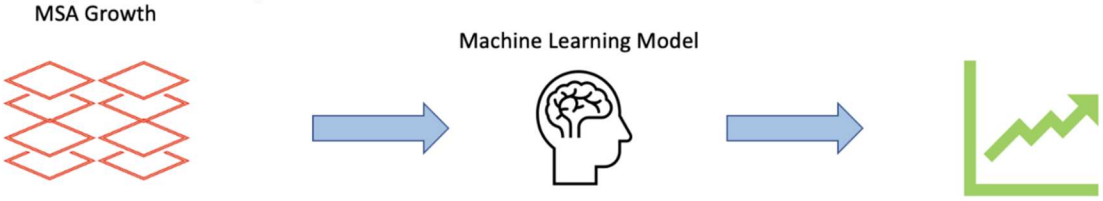

Figure 18: System Resource Planning Model

Conclusion

This article extensively examines the key aspects of machine learning and their integration into microservices architecture. The authors analyze various design patterns in the context of microservices architecture, such as SAGA, CQRS, API Gateway, and CircuitBreaker, identifying their advantages and drawbacks.

Research is conducted on various classes of machine learning (ML) models, from regression to multiclass classification models and models for time series analysis, as well as modern libraries for creating Microservices (MSA) models using the Python programming language.

Based on the analysis results of MSA and ML elements and approaches, it is concluded that integrating ML into MSA is an important step to enhance the efficiency and capabilities of complex and large-scale systems. Such an approach allows for automating processes, improving decision-making quality, and responding to changes in real-time. Overall, this provides a well-founded understanding of the interaction between machine learning and microservices, as well as offering practical recommendations and examples for successful integration of these technologies into modern systems.

The material presented in this research on the interaction of machine learning and Microservices provides a valuable foundation for further investigations by authors in the field of intelligent web service scaling systems based on MSA using ML.

References

[1] E. Zharikov, S. Telenyk, O. Rolik, Method of Distributed Two-Level Storage System Management in a Data Center Advances in Intelligent Systems and Computing, 938 (2020) 301–315. DOI:10.1007/978-3-030-16621-2_28.

[2] P. Raj, A. Raman, H. Subramanian, Architectural Patterns. Packt Publishing (2017). ISBN: 9781787287495.

[3] S.Newman, Building microservices: Designing fine-grained systems. Beijing i pozostałe: O'Reilly, 2021. ISBN: 978-1492034025.

[4] M. Bruce, P. Pereira, Microservices in action. Shelter Island, NY: Manning Publications Co., 2019. ISBN: 9781617294457.

[5] A. Müller, S. Guido, Introduction to machine learning with python: A guide for data scientists. Sebastopol: O'Reilly Media, 2018. ISBN: 978-1-449-36941-5.
[6] J. Mueller, Machine learning security principles: Use various methods to keep data, networks, users, and applications safe from Prying eyes. Birmingham: Packt Publishing, 2023. ISBN: 978-1-80461-885-1.
[7] S. Raschka, Y. Liu, and V. Mirjalili. Machine learning with pytorchand Scikit-Learn: Develop machine learning and deep learning models with python. Birmingham: Packt Publishing, 2022. ISBN: 978-1-80181-931-2.
[8] M. Abouahmed, and O. Ahmed. Machine learning in microservices: Productionizing Microservices Architecture for Machine Learning Solutions. Birmingham: Packet Publishing, 2023. ISBN: 978-1-80461-774-8.
[9] Ubuntu server - for scale out workloads Ubuntu, 2023. URL: https://ubuntu.com/server/
[10] A. Kupin, Y. Osadchuk, R. Ivchenko, O. Gradovoy. The Methods for Training Technological Multilayered Neural Network Structures (2021), in: CEUR Workshop Proceedings, 3013, pp. 327–333. URL: https://ceur-ws.org/Vol-3013/20210327.pdf.
[11] J. Brains, PyCharm: The python IDE for professional developers by jetbrains, JetBrains, 2021. URL: https://www.jetbrains.com/pycharm/
[12] DBeaver Community, 2023. URL: https://dbeaver.io/
[13] MySQL, 2023. URL: https://www.mysql.com/
[14] Accelerated Container Application Development, 2023 Docker. URL: https://www.docker.com/