Neural Network Technology for Real-Time IT Service Management

Viktor Krasnoproshin
Faculty of Applied Mathematics and Computer Science
Belarussian State University
Minsk, Belarus
Email: krasnoproshin@bsu.by

Aleksandr Starovoitov
Faculty of Applied Mathematics and Computer Science
Belarussian State University
Minsk, Belarus
Email: StarovoytovAA@bsu.by

***Abstract*—The paper explores a relevant applied problem related to building decision-making systems for resource management of critical IT services. The uncertainty of external load is an important factor that affects operational management. Neural network forecasting is used to improve control systems. A model system of a critical IT service is described, an original technology, structure and architecture of the control system are proposed. Experiments have been conducted to confirm the workability of the proposed technology.**

***Keywords*—decision-making, information system, proactive management, uncertainty of external load, neural networks, critical IT service**

I. Introduction

Support for computing systems in critical infrastructures (such as banking, telecommunications, industrial systems) in an operational state (with guaranteed computational resource levels) is a relevant problem in today's digital society.

Uncertainty in external loads and outages of computing equipment lead to operational failures and performance degradations of critical IT systems. As a result, the loss of operational efficiency in processing information and conducting banking and other operations can have serious consequences, including financial losses and major incidents.

Making operational decisions for the management of critical IT services allows for the reduction or prevention of negative consequences. However, the human factor often contributes to a decrease in operational efficiency. Therefore, various automated solutions are actively being developed to enhance efficiency and proactivity in the management of critical systems.

In laboratory conditions, it is difficult to develop relevant systems (without interacting with the critical infrastructure itself). Therefore, one of the current problems is the creation of model systems that enable researchers to use them for the development of proactive management systems for critical IT services.

Several authors develop various management systems for critical IT services using neural network models, which contribute to efficient decision-making [1]–[4]. However, in these systems, only one neural network model is trained for specific types of external load. It is assumed that these models will successfully forecast the values of necessary parameters for other types of load associated with uncertainty. In these works, training datasets are prepared in advance, containing long time series with a large number of elements. Training on such datasets takes a considerable amount of time and requires high-performance resources (GPU, TPU) to effectively train models within an acceptable timeframe.

This paper considers the construction of a model system conceptually corresponding to a critical IT service and an operational management system with a neural module.

A multi-model approach is used, based on the idea that the managed IT system can be in various states (in terms of computational resource volume), and for each state during its existence, a neural network model can be created and trained to predict the average %CPU utilization across computational modules.

A combined management system is used, in which the control decision for state changes is formed based on both reactive and proactive approaches.

II. Critical IT Service: Conceptual Model

Let's consider the conceptual model of a critical information system, which describes its basic elements and significant parameters. It can be described in the form of the following tuple:

$$\text{System} = (\text{Clients} \cup \text{Balancers} \cup \text{APPs} \cup \text{DBs}, \text{Links}), \quad (1)$$

where

$\text{Clients} = \{\text{Client}_i\}$ — the set of system clients. These can be both user devices and other external systems;

$\text{Balancers} = \{\text{Balancers}_j\}$ — the set of balancers that distribute requests from clients and external systems across services. Services within the system can also communicate with each other through balancers;

APPs $= \{\text{APP}_k\}$ – the set of application services within the system;

DBs $= \{\text{DB}_l\}$ — the databases of the system (can be of any type);

Links $= \{\text{Link}_m\}$ — the set of bidirectional temporal links that arise during communication between elements of the system;

The elements Client_i, Balancers_j, APP_k, DB_l, Link_m can also be represented as tuples:

$\text{Client}_i = (\text{Protocol}_i, \text{Profile}_i)$, where Protocol_i defines the specification, and Profile_i represents the interaction profile (requests, their parameters, frequency, etc.) for the i^{th} client;

$\text{Balancers}_j = (\text{Protocol}_j, \text{RPS}_j, \text{Throughput}_j)$, where RPS_j is the number of requests per second, and Throughput_j is the throughput (Mbps) for the j^{th} balancer. It is assumed that the balancer supports all interaction protocols, and the delay on the balancer is insignificant compared to the response time;

$\text{APP}_k = (\text{Compute\_Modules}_k, \text{Soft\_Service}_k)$ the set of instances of application software, where Compute_Modules_k is the set of computational modules, and Soft_Service_k is the type of application service used on the computational modules Compute_Modules_k;

$\text{DB}_l = (\text{Compute\_Modules}_l, \text{Soft\_Service\_DB}_l)$ — describes a set of database instances serving one type of application service instances, where Compute_Modules_l is the set of computational modules, and Soft_Service_DB_l is the type of database software used in the computational modules Compute_Modules_l. These can be any (not necessarily relational) databases;

$\text{Link}_m = \{(\text{P}_{mn}, \text{P}_{mo})\}$ — the set of bidirectional temporal links between elements of the system, established through communication via specific open ports;

Compute_Modules $= \{\text{Compute\_Module}_c\}$ — the set of computational modules of the system, where each computational module can also be described by the following tuple:

Compute_Module = (CM_Type,
CM_CPU_Limit, CM_CPU_Perf,
CM_RAM_Limit, CM_RAM_Perf,
CM_Storages, CM_NET_Throughput),
where

- CM_Type $\in$ {"physical server", "logical or hw partition", "virtual machine", "container"} — the type of module;
- CM_CPU_Limit — the number of processors (CPU or vCPU) available to the module;
- CM_CPU_Perf — the maximum performance of one processor in the module;
- CM_RAM_Limit — the available volume of RAM in the module (GiB);
- CM_RAM_Perf — the maximum performance of RAM in the module (determined by bandwidth and memory access time);
- CM_NET_Throughput — the maximum throughput capacity of the module (Gbps).

CM_Storages $= \{\text{CM\_Storage}_p\}$ — storage modules available to the computational module. Each storage module is defined by a tuple:

CM_Storage = (CM_Storage_Type,
CM_Storage_Capacity, CM_Storage_Perf), where

- CM_Storage_Type $\in$ {"local", "external"} — determines the type of connection of the storage module to the computational module. In this case, the local option describes local disk resources (HDD, SSD). The "external" component refers to storage resources external to the computational module. These can be connected using various input-output devices that support different block protocols (iSCSI - SCSI over Ethernet, Fiber Channel Protocol – SCSI over Fiber Channel, NVMe-oF – NVMe over Fiber Channel), file protocols (CIFS, NFS) and object protocol (S3).
- CM_Storage_Capacity — the storage capacity of the module (GiB).
- CM_Storage_Perf — the performance of the module can be defined as the number of Input/Output Operations Per Second (IOPS) (with minimum response time) or throughput (GBps), for different workload profiles.

The workload profile is determined by the percentage of read operations (%Read), the size of data blocks (KiB), and the distribution, which indicates the presence of block sizes in requests, their proportion, and the delays associated with them in the total stream of requests.

Soft_Service $= \{\text{Soft\_Service}_k\}$ — types of application services in the system used by computational modules in the system.

Soft_Service_DB $= \{\text{Soft\_Service\_DB}_l\}$ — types of database software used in computational modules.

In information systems, for integrating various services, elements such as message brokers (e. g., RabbitMQ, IBM WebSphere MQ, ActiveMQ Artemis, Apache Kafka, etc.) are often used. In our case, these services are not allocated to a separate class since essentially they can be attributed either to the set of entities in APPs or to DBs if this functionality is implemented at the database level (e. g., Oracle Advanced Queuing).

The model structure explicitly does not include: telecommunication equipment (Switches, Routers), resource management systems, and various perimeter control systems of the information system. All of these may restrict access to the system from outside and between components based on ports and protocols (Firewalls). Additionally, they may lead to deeper inspection of the exchange at the application level (WAF — Web Application Firewalls) and the necessity to perform analysis of exchanges at OSI Model layers 3 and 4 (IPS — Intrusion Prevention System), tracking anomalies and conducting

checks for known vulnerabilities and attack vectors based on signature databases and established policies.

The structure also explicitly does not include monitoring and logging systems, antivirus protection, encryption, as well as other technological services and systems (LDAP, DNS, backup and recovery services, CI/CD systems, time synchronization service, etc.). It is assumed that these services are properly configured, and their operation does not affect the functionality of the system being considered.

III. Model System: Task Statement

With consideration of the described model, the following task is formulated:

To develop the system that conceptually corresponds to the previously described model of a critical information system (1), meeting the following criteria:

- The system can operate on a workstation, laptop, or virtual machine with resources not exceeding 4 CPU cores and 8 GiB of RAM;
- The set of APPs is represented by a compact web application capable of handling external requests;
- An instance of the web application service operates within the computational module Compute_Module, which has a specific limit on CPU resources (other limits are also possible but not mandatory);
- The system has the capability to scale within the specified resources mentioned above, meaning the number of instances of the web application service can be adjusted during operation under load by increasing or decreasing the number of computational modules. Scaling management is available both programmatically and manually;
- For each computational module of the system CPU utilization metrics are collected, with a metric collection period $\sim 2s$. The data is stored in CSV files with the specified frequency. When additional computational modules are added, statistics collection is automatically enabled for them;
- The system should have a reactive scaling module that operates according to the following algorithm: after receiving metrics of %CPU utilization for the running computational modules hosting an application web service, the average utilization percentage $\overline{\%\text{CPU}}_N$ is calculated based on the number (N) of running modules. If $\overline{\%\text{CPU}}_N > 50\%$, the system automatically adds another computational module with the application web service. If $\overline{\%\text{CPU}}_N < 20\%$, the system automatically shuts down one computational module with the application web service. If $\overline{\%\text{CPU}}_N$ is in the range from 20% to 50%, the system does not perform any configuration changes;
- After each change in the system's state (during scaling), a stabilization mode must be activated, during which the reactive control system does not perform any configuration changes to the application web service for a specified period of time;
- The set of Balancers is represented by a request balancing service between instances of the web application;
- When a new instance of the web application is added, it is automatically included in the load balancing. Similarly, when an instance of the web application is turned off, it is automatically excluded from the load balancing;
- The set of Clients is represented by a load testing system where you can define a workload profile for the application system based on requests and various user profiles. The workload profile can be defined using a function, pre-prepared data, or through the operation of web and CLI clients. The load testing system should display real-time statistics during testing and have the capability to save reports containing statistics on specific requests such as RPS, Response Time, errors during request execution, and the number of users;
- The load testing system operates within the computational module and can support a distributed configuration of instances running simultaneously across multiple computational modules;
- During a load test with maximum load, the CPU resource consumption by the computational module, where the load testing system operates, should not exceed the capacity of one CPU core;
- The presence of a database instance is possible but not mandatory;
- The system supports the Infrastructure as Code (IaC) model.

IV. Model System: Implementation

The algorithm used to solve the given task includes the following main stages:

- Defining the key functional blocks of the system. Preparation of a high-level schematic diagram;
- Identifying possible implementation options for each block considering the system requirements;
- Analyzing possible implementations considering the following criteria: availability of ready-made components that can be used to build functional blocks, simplicity of implementation, and implementation time;
- Choosing an implementation option for prototyping the system;
- Creating a prototype of the system;
- Qualitative assessment of the prototype's compliance with the task criteria during testing;
- If the criteria are not met, return to step 2;
- Perform the necessary number of iterations (steps 2-6) until the prototype meets the task criteria.

Next, the functional blocks of the system were defined and a high-level schematic of the prototype was constructed (Fig. 1):

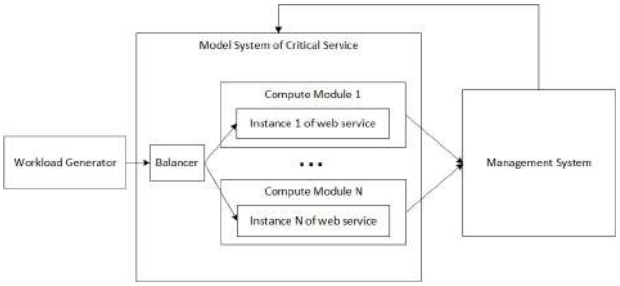

Figure 1. Structural block diagram of Model Critical Service.

A. Computational Modules

This functionality is core as it forms the basis for various solutions. In works [1]–[4], solutions for proactive management tasks were based on virtual machines deployed in various public clouds (such as Amazon, Google, etc.) or Private Clouds or Data Centers. For managing virtual machines (creation, launch, stop, deletion), cloud service capabilities or data center management systems were used. Virtual machines provide good application isolation but require more computational resources since each virtual machine needs resources for the operating system. Additionally, it is necessary to use a module that replicates the functionality of a cloud service for managing virtual machines (creation, launch, stop, deletion), as well as perform configuration of operating systems in virtual machines, installation, configuration, and management of the application web service, and support the Infrastructure as Code (IaC) model.

As a result of the research, the decision was made to use containers as the computational modules. They are less resource-intensive, simpler to implement, and meet the criteria III.

Several solutions were considered: Docker Compose [5], Kubernetes Cluster [6]. The Docker Compose option turned out to be simpler, although the use of the Kubernetes Cluster with a single worker node (e. g., Minikube [7]) is also possible.

As a result, the decision was made to use Docker Compose for the computational module block in the system prototype. This solution allows for the creation of computational modules (Docker containers), their management, and the collection of container utilization metrics. There are ready-made libraries (e. g., Docker SDK for Python [8]) for working with the Docker API Engine.

Load balancing across containers with web application is achieved using Docker Compose's built-in features based on the service name. The configuration of services and resources is described in a YAML file.

B. Web Application Service

To conserve resources, ensure stability, and simplify setup and operation, it was decided to implement the web application as a microservice based on the popular high-performance minimalist web framework echo.labstack [9], written in the high-level language Go [10].

C. Load Generator

Two options of load generation software were considered: Apache JMeter [11], written in Java, and Locust [12], written in Python. Apache JMeter is more feature-rich, complex to configure, and resource-intensive, whereas Locust has less functionality, is easier to set up, allows load profiles to be described as Python classes, is less resource-intensive, supports distributed instance configuration, and allows defining user load profiles as functions or pre-prepared data. Locust was chosen as the load generator for the prototype.

During debugging, it was discovered that in the case of a large number of lightweight requests, the CPU utilization of the container running Locust noticeably exceeds the CPU utilization of containers hosting the web application services. It was necessary to add an additional endpoint (/load) at the web application level, which invokes Go code. This code, using parallel goroutines, achieves the desired increase in CPU utilization.

The addition of this more CPU-intensive request allowed the rebalancing of the resource utilization between the load generator and the system under test. Experimentally, it was determined that the prototype system, with the resource limit specified in criterion III and the generated load profile, can handle up to 700-800 active users. Further increasing the load leads to reaching the limit of resource utilization on the virtual machine hosting Docker Compose.

Next, we will discuss the principles of operation and implementation of a combined control system with a neural module.

V. Combined Control System: Operation Principle

To manage the resources of a critical IT system, an agent is used, which in real-time receives utilization data from computational modules and makes decisions regarding the scaling of the managed system. The agent employs a combination of reactive and proactive management. For each state of the managed system, a unique initial dataset is automatically generated, a neural network model is trained based on this dataset, and predictions of resource utilization parameters are made.

The agent compares the current data of average load across computational modules with the forecast results for a specific system state and makes decisions regarding state changes (scaling).

If there is a sharp peak in load and the neural network model is not yet ready or there is no forecast for the

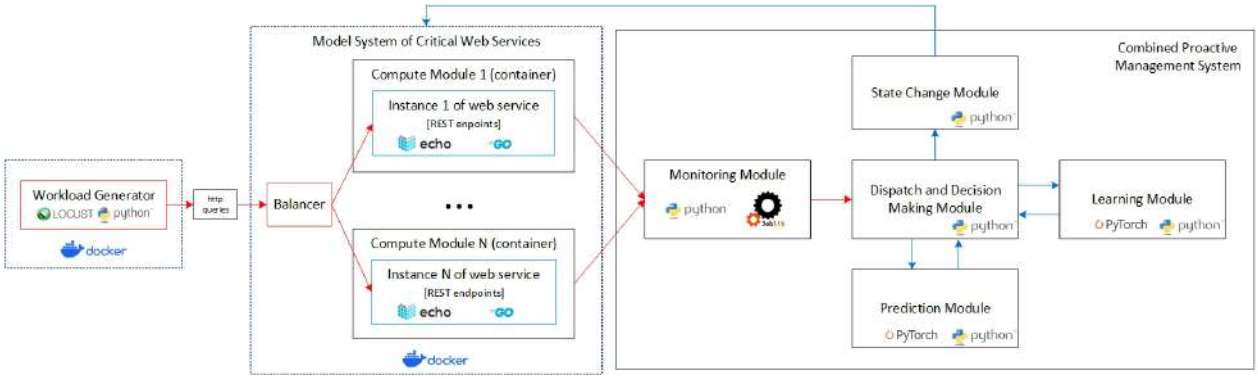

Figure 2. Structural block diagram of the combined control system.

average utilization of modules, or if such behavior is not included in the forecast, the reactive component is triggered.

If the forecast of average parameters exceeds threshold values, a proactive decision is made in advance regarding the state change of the system.

VI. Combined Control System: Implementation

The system architecturally consists of 5 main modules, which are depicted in (Fig. 2).

A. Monitoring Module

The monitoring module is responsible for collecting performance metrics from the computational modules of the system and adding/removing new metrics (when the system state changes). It is worth noting separately that when working with the Docker API Engine, there is a peculiarity related to the fact that the API does not return statistics for all containers in a single request, as the docker stats console command does. Additionally, the request itself takes $\sim 1s$ to execute because the Docker daemon needs a certain interval to calculate the corresponding average metric values. Therefore, the joblib library [13] was used for correct metric collection, which allows for implementing parallel requests.

B. State Change Module

The state change module is responsible for sending control commands (which modify the state of the managed system) and ensuring the correctness of state changes.

C. Dispatch and Decision-Making Module

The dispatch and decision-making module is central. It and the other modules are implemented in the high-level language Python. More detailed algorithm of its operation is presented in Figure 3 (see [16] for more details).

During initialization, resource utilization thresholds (system SLAs) are set, reaching which leads to a change in the state of the managed system. Separate thresholds are set for adding (A) and removing (D) computational resources. A system stabilization parameter (cool_period) is set, determining the number of cycles during which no state changes are performed in the system. A parameter is set to determine the number of data accumulation iterations for one state to create a model (M). A lead time parameter (Z) is set – the number of forecast points into the future.

In the main process, after the initialization stage, a loop is implemented in which each iteration involves refining the composition of computational modules of the managed system, obtaining current values of utilization of computational modules, and making decisions regarding the change of the managed system's state.

Metrics are collected with a period $\sim 2s$. Historical data on the utilization of each computational module for different system states are saved as CSV files. Data on module utilization for the current state are accumulated in memory in a dictionary. Using the latest collected data, the main process calculates the average value across the set of computational modules (R).

If R exceeds the threshold for addition (A), a decision is made to add resources to the managed system. If R is less than the removal threshold (D), a decision is made to remove resources. After the decision is made, a command is sent to the state change module (reactive component).

In this process, mechanisms of non-blocking interaction with other processes are implemented, which run in parallel with the main process and responsible for creating and training neural networks and forecasting utilization parameters for a specific state with a given lead time. This mechanism is implemented using the multiprocessing [14] library. Inter-process communication uses the multiprocessing.Queue mechanism, which forms FIFO (first input first output) queues.

Three queues (model_list_q, model_state_q, model_result_q) are used for interaction between the main process and the process responsible for creating and training the neural network model, and three queues

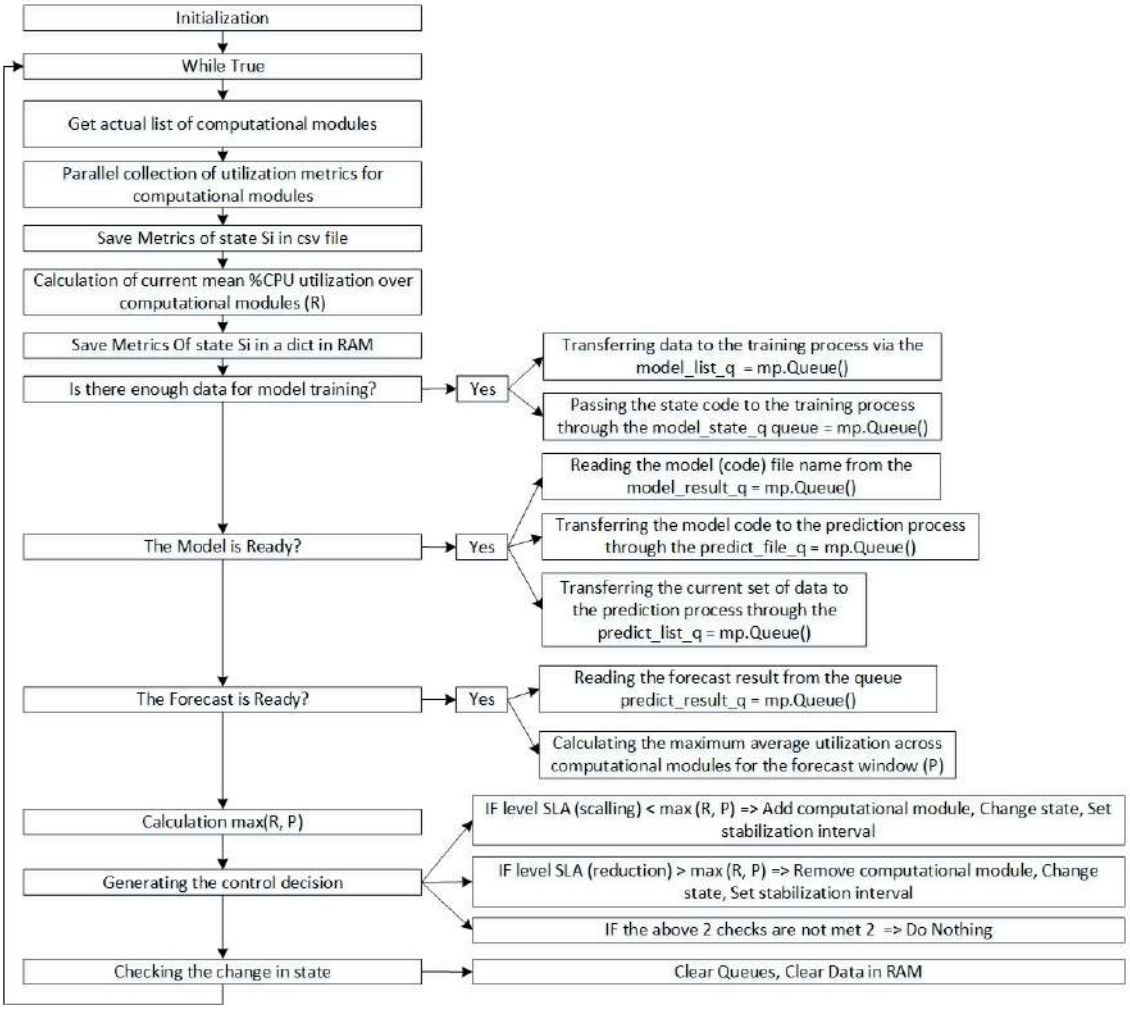

Figure 3. Control system operation algorithm.

(predict_file_q, predict_list_q, predict_result_q) are used for interaction with the prediction process (see Fig. 4).

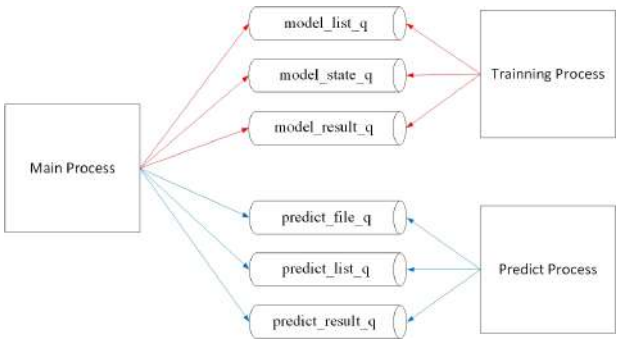

Figure 4. Multiprocessing queues.

D. Training and Prediction Modules

Both modules are implemented in Python, the PyTorch [15] library is used. The training module creates neural network models based on datasets for various states of the managed system and saves the models in the neural network model library. Each model is encoded with state index.

The prediction module loads models by state index from a library of neural network models and performs utilization forecasts with a certain advance for different states of the controlled system.

VII. Parameters of Neural Networks

We assume that each subsequent element in the time series depends on a certain number of previous elements of the series, i. e., lagged values of the original series are used as independent parameters of autoregression. The number of such parameters determines the moving window for the time series. We choose the window size (tw) to be 30 elements. (determined empirically).

We assume that the primary contribution to the approximating function for the next element in the time series comes from a combination of the previous 30

elements of the series. A neural network with one hidden layer and one output neuron is used. The number of neurons in the hidden layer (90) is three times larger than the size of the time window.

In the output layer, there is 1 neuron acting as a summation unit. A fully connected linear layer (nn.Linear) is used with the ReLU activation function. For regularization, neuron dropout is used with a dropout probability of 0.015. The mean squared error (MSE) loss function (nn.MSELoss()) is used. The optimization method used is torch.optim.Adam with a learning rate of 0.002. Preprocessing of the original series with scaling to the interval $[0, 1]$ is not performed because it introduces additional error into the raw data and leads to additional overhead costs for scaling before and after training.

A short time series of 64 samples is used. The interval between samples is $2s$. The duration of the series is $128s$. The original series is divided into two datasets — train (70% - 43 samples) and test (30% — 21 samples). Considering that tw $= 30$, the number of examples for training is 14. Training is conducted for 100 epochs. The batch size is set to 1. The training time for the neural network with the specified architecture on the dataset $\sim 2s$. The prediction execution time, with forecasting future 40 samples $\ll 1s$.

VIII. Description of NN Models Usage Process in the Control System

After obtaining the model for the current state, the main process transfers information about the model to the prediction process. The name of the model file is passed to the predict_file_q queue, and the current utilization data set for forecasting is passed to the predict_list_q queue. The prediction process checks for the presence of data in the specified queues at intervals corresponding to the data collection frequency. After reading the data from the queues, the prediction is executed, and the forecasted data is passed to the model_result_q queue. After receiving the forecast results for the current state, the main process calculates the maximum average utilization (P) across computational modules for the forecast window (Z).

Next, $max(R, P)$ is computed – the maximum between the current utilization value and the maximum forecasted value. This value, together with the code of the current state, the number of computational modules, and the stabilization limit, is used to make a decision about scaling the system. At the same time, if $max(R, P)$ exceeds the addition threshold (A), a decision is made to add resources to the managed system. If $max(R, P)$ is less than the removal threshold (D), a decision is made to remove resources. After the decision is made, a command is sent to the state change block.

IX. Example of the System Under Workload

As an example, Figure 5 illustrates the operation of the system under workload, gradually increasing to 600 users performing various requests to the system over approximately 0.5 hours.

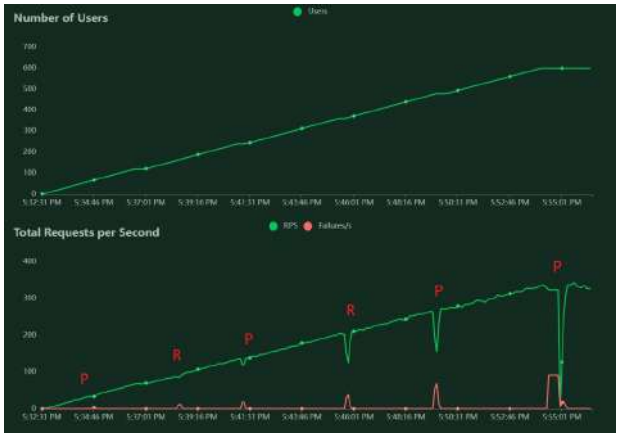

Figure 5. Model System Workload (Number of Users and RPS).

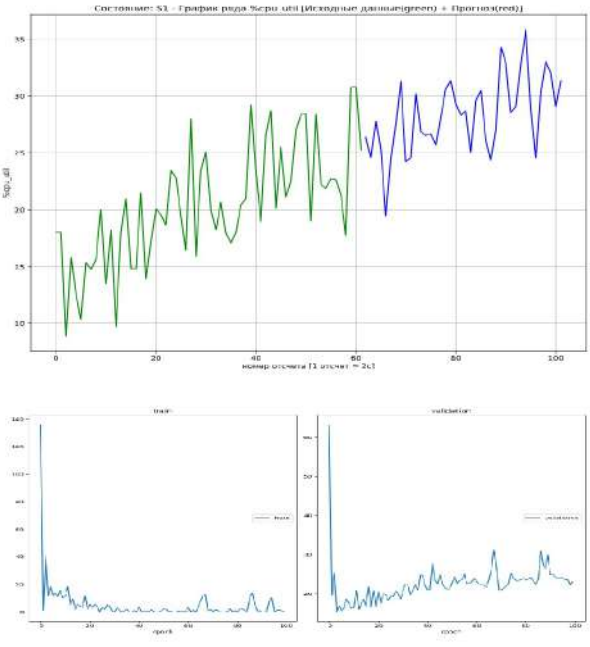

Figure 6. Forecast 1 for state S1.

As a result of the run, the system changed its state 6 times, with 4 changes being proactive and 2 being reactive.

Figure 6 shows an example of utilization forecast for computational modules in state S1 (green color represents the original data, blue color represents the forecast).

Figure 7 illustrates the situation with a reactive state change (sharp peak around the 112th sample). In this

case, the neural network model did not win, although it predicted the state change threshold exceedance (50%).

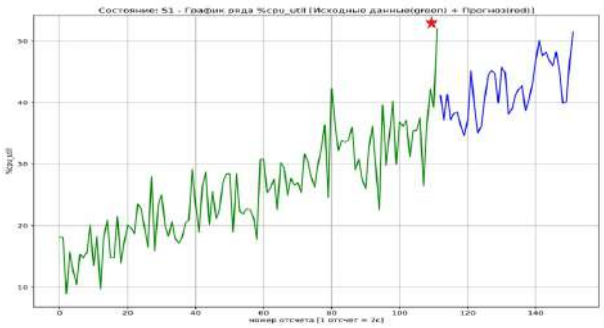

Figure 7. Forecast 51 for state S1.

This situation shows that the complexity of the process increases as the number of users (the number of requests to the system). The neural model does not account for the change in complexity, as the complexity (architecture) of the model itself does not change.

X. Results

The problem related with making real-time decisions in managing the computational resources of a critical IT service under conditions of uncertainty in external load was discussed in the article. As a result of the conducted research, the model system of critical IT service was developed. The architecture of the combined management system are proposed. The technology for real-time decision-making is developed and described, which has been implemented in practice. Experimental researches have been conducted, confirming the workability of the proposed technology.

The original multi-model approach to forecasting problem in case of generating control decisions is proposed, which enhances the adaptive properties and stability of the managed system to external workload. This approach allows to create a library of neural models capable of making forecasts needed parameters for various system states. It becomes possible to further complicate the predictor through the use of ensembles of models. At the same time, a more complex architecture (potentially capable of more accurate predictions) requires increased training time, which reduces the speed of decision making. This situation clearly shows that fast decision-making with preparing a more accurate forecast for a complex signal requires not only an improvement in the algorithm but also more computing resources with high performance.

References

[1] M. Straesser, J. Grohmann, J. von Kistowski, S. Eismann, A. Bauer, S. Kounev. Why Is It Not Solved Yet?: Challenges for Production-Ready Autoscaling // In ICPE '22: ACM/SPEC International Conference on Performance Engineering, Bejing, China, April 9 - 13, 2022, P. 105–115, ACM, 2022.

[2] N. Khan, D. A. Elizondo, L. Deka, M. A. M.–Cabello. Fuzzy Logic applied to System Monitors // IEEE Access, Vol. 9, P. 56523-56538, 2021.

[3] B. M. Nguyen, G. Nguyen, Giang. A Proactive Cloud Scaling Model Based on Fuzzy Time Series and SLA Awareness // Procedia Computer Science (International Conference on Computational Science ICCS 2017), 108, P.365–374, 2017.

[4] V. Persico, D. Grimaldi, A. Pescape, A. Salvi, S. Santini. A Fuzzy Approach Based on Heterogeneous Metrics for Scaling Out Public Clouds. IEEE Transactions on Parallel and Distributed Systems, Vol. 28, No. 8, P. 2117–2130, 2017.

[5] Docker Compose Overview [Elektronnyy resurs]. Rezhim dostupa: https://docs.docker.com/compose/. Data dostupa: 25.03.2024.

[6] Kubernetes Cluster [Elektronnyy resurs].Rezhim dostupa: https://kubernetes.io/. Data dostupa: 25.03.2024.

[7] Minikube Local Kubernetes Cluster [Elektronnyy resurs]. Rezhim dostupa: https://github.com/kubernetes/minikube/. Data dostupa: 25.03.2024.

[8] Docker SDK for Python [Elektronnyy resurs]. Rezhim dostupa: https://github.com/docker/docker-py/. Data dostupa: 25.03.2024.

[9] Echo LabStack High performance extensible minimalist Go web framework [Elektronnyy resurs]. Rezhim dostupa: https://echo.labstack.com/. Data dostupa: 25.03.2024.

[10] Go an open-source programming language supported by Google [Elektronnyy resurs]. Rezhim dostupa: https://go.dev/. Data dostupa: 25.03.2024.

[11] Apache JMeter [Elektronnyy resurs]. Rezhim dostupa: https://jmeter.apache.org/. Data dostupa: 25.03.2024.

[12] Locust [Elektronnyy resurs]. Rezhim dostupa: https://locust.io/. Data dostupa: 25.03.2024.

[13] Joblib Python library [Elektronnyy resurs]. Rezhim dostupa: https://joblib.readthedocs.io/en/latest/parallel.html/. Data dostupa: 25.03.2024.

[14] Multiprocessing Python library [Elektronnyy resurs]. Rezhim dostupa: https://docs.python.org/3/library/multiprocessing.html/. Data dostupa: 25.03.2024.

[15] Pytorch [Elektronnyy resurs]. Rezhim dostupa: https://pytorch.org/. Data dostupa: 25.03.2024.

[16] Starovoytov, A.A. Algoritm proaktivnogo upravleniya vychislitelnymi resursami. 80-ya nauchnaya konferentsiya studentov i aspirantov Belorusskogo gosudarstvennogo universiteta [Elektronnyy resurs. Materialy konf., Minsk, 10–20 marta 2023 g. V. 3 ch. Ch. 1. Belorus. gos. un-t, redkol.: A. V. Blokhin (gl. red.) [i dr.], Minsk, BGU, 2023, 398 s.

НЕЙРОСЕТЕВАЯ ТЕХНОЛОГИЯ ОПЕРАТИВНОГО УПРАВЛЕНИЯ ИТ СЕРВИСОМ

Краснопрошин В. В., Старовойтов А. А.

В работе иссследуется актуальная прикладная проблема, связанная с созданием систем принятия оперативных решений для управления ресурсами критически важных ИТ сервисов. Неопределенность внешней нагрузки является важным фактором, влияющим на оперативное управление. Для улучшения работы систем управления предлагается подход на основе мультимодельного нейросетевого прогнозирования. Описана модельная система критично ИТ сервиса. Предложена оригинальная технология, структура и архитектура системы управления. Проведены эксперименты, которые подтвердили работоспособность указанной технологии.

Received 13.03.2024