Debugging Cloud Continuum Blueprint Primitives with an ML-based Steering Method Toward Extreme Conditions

Robert Lovas

Institute for Computer Science and Control (SZTAKI), Hungarian Research Network (HUN-REN), Kende u. 13-17, H-1111 Budapest, Hungary; robert.lovas@sztaki.hun-ren.hu

John von Neumann Faculty of Informatics, Óbuda University, Bécsi út 96/B, H-1034 Budapest, Hungary; lovas.robert@nik.uni-obuda.hu

Abstract: Debugging high-dimensional state spaces in cloud continuum environments poses significant challenges, particularly when investigating extreme conditions such as high latency, competing on resources, or configuration anomalies. This paper presents a novel supervised machine learning-based approach to efficiently assist the debugging process by steering toward potential fault states in an automated way. Leveraging typical blueprint primitives, such as load balancers and temporal data storage in the presented case studies, Multi-Layer Perceptron (MLP) and Dense Neural Networks (DNN) were trained to predict the distance to extreme situations. The trained model informs a traversal mechanism that explores the state space using this heuristic, minimizing the time and consumed resources required to detect actual faults. The first experiments conducted with two foundational blueprint primitives (buffers and multi-tier load balancers) demonstrate the promising effectiveness of the approach in locating potential fault states. By integrating this method into cloud-edge debugging tools, developers can enhance not only fault localization but reliability and performance as well, particularly for extreme timing conditions. Future work will explore a wider set of primitives, as well as adjacency matrix representations and convolutional techniques, to improve applicability, scalability and robustness of the presented solution.

Keywords: Cloud computing; Debugging; Machine Learning; Fault Detection; Markov chains; State Space Explorations

1 Introduction and Background

Cloud blueprint primitives [1] serve as foundational components in orchestrating cloud-based systems, enabling (among others) seamless deployment and scaling of platform or applications. Temporal data storage and load balancers [2] play crucial

roles in the cloud-edge continuum [3] as blueprint primitives due to the widely applied data stream-oriented processing mechanisms. However, extreme conditions such as high latency, competing on resources, or configuration anomalies can disrupt these primitives, often decreasing system reliability and performance as well. Localization of related faults is a cumbersome process [4]. This article proposes a smart steering mechanism designed to debug cloud blueprint primitives under such extreme timing conditions. During the modelling phase I focused on the conditions where a component is starving (e.g. there is no data in the buffer), or overloaded (e.g. the data buffer is full).

The presented work builds upon our previous study [5], where we analyzed debugging issues concerning the Producer-Consumer primitive, used as our baseline model. We extended the scope to include more complex topologies with load balancers (advanced use cases involving multiple Producers and/or Consumers) and state-of-the-art service meshes.

In our previous work, we proposed an advanced, ML-enhanced debugging framework for cloud services (see Figure 1). The selected modeling toolset — including the PRISM language, simulator, and model checker [6] — has proven effective for handling relatively large transition graphs and state spaces. It also enabled the generation of training datasets by leveraging the formal descriptions of Markov Chains and Decision Processes.

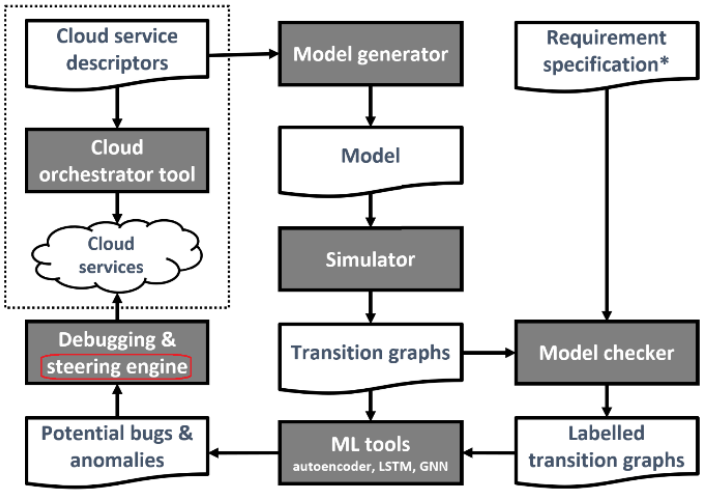

Figure 1
Proposed framework for smart debugging with formal modelling and ML tools [5]

We successfully conducted autoencoder-supported anomaly (fault) detection experiments on Producer-Consumer primitive-based topologies of gradually increasing complexity (see ML tools on Figure 1). These experiments provide a foundation for steering mechanisms for cloud debuggers featuring replay and active

control mechanisms. Our Graph Neural Network (GNN) based fault detection also produced promising results. However, experiments using an advanced steering mechanism based on Long Short-Term Memory (LSTM) did not lead to satisfactory results.

In this article, I show a strategy to overcome the limitations of our previous work [5] and I present a method based on supervised machine learning principles that can guide the debugger tool toward potential fault states within a high-dimensional state space, see the highlighted 'steering engine' on Figure 1. This novel method aids in selecting states for further debugging and testing where the possibility of discovering faults is higher. This procedure is particularly useful when the state space is too large to exhaustively traverse and test all possible states individually. The proposed Multi-Layer Perceptron (MLP) and Dense Neural Network (DNN) model-based methods present an approach that can effectively tackle this problem. Through a selected example dataset, I demonstrate how this procedure can be applied in practice.

The steering mechanism is to be incorporated into debuggers that support advanced active control and replay techniques, including our DIWIDE debugger for parallel and distributed applications [7], or other cloud debuggers designed for Terraform cloud orchestrator [8] [9] or for microservice meshes [5].

The paper is structured as follows. In this section, I introduced and outlined the problem, providing justification for the relevance of my approach and some important background information on our previous results in debugging complex parallel and distributed systems. Section 2 provides a review on research findings and methods that overlap with and/or relate to the results presented in this paper. In Section 3, I present the methodology used to address the problem, explaining its steps in detail, and how they contribute to solving the addressed problem. Section 4 showcases the results achieved on two basic but crucial test scenarios using the proposed method. The paper concludes with Section 5.

2 Related Work

The presented methods in this section particularly lack a systematic approach for guided traversal toward potential fault state. In contrast, our work combines Markov chain modeling with supervised ML-based heuristic steering, enabling efficient exploration and efficient fault localization in high-dimensional state spaces for cloud-edge debuggers.

2.1 ML-based Log and Time-Series Analysis for Anomaly Detection

Gan et al. [10] introduced SAGE, an unsupervised ML-driven framework for performance debugging that is effectively scalable for inspecting microservice architectures. However, their focus is primarily on performance debugging related to Quality-of-Service (QoS) violations, without a formal modeling background.

In another related work, Huang et al. [11] proposed log-based anomaly detection methods to troubleshoot faults in complex cloud systems. Their method automates log parsing to identify anomalies, focusing primarily on root cause analysis but lacking predictive fault distance estimation and active steering.

Shahane [12] outlined the use of both supervised and unsupervised ML for not only automated failure detection but also anomaly prediction in cloud infrastructure. Their high-level approach aligns with our work in many aspects; however, the author only presents a conceptual framework without concrete implementation.

Additionally, Yu et al. [13] elaborated MLPing, a proactive fault detection system for large-scale distributed networks. While MLPing focuses on anomaly detection and real-time alerting, our approach extends fault detection by incorporating heuristic steering in high-dimensional state spaces.

Han et al. [14] proposed FRAPpuccino, a runtime fault detection framework with functionalities to capture a comprehensive view of program activities at the Platform as a Service level using a directed acyclic graph (DAG) representation. Its dynamic sliding window algorithm, combined with clustering, proves effective for runtime anomaly detection. However, it lacks an active and supervised ML-based steering mechanism for navigating extreme situations.

Kumar et al. [15] introduced LSTM models for early fault detection through system log analysis. While this method effectively identifies anomalies in time-series data, it does not integrate probabilistic modeling or a steering mechanism, as demonstrated in our approach.

2.2 Markovian Methods

Cao and Niu [16] explored higher-order Markov graphs for fault detection, specifically analyzing cloud deployments and big data logs to detect anomalies. Their method proves effective for identifying complex patterns; however, it lacks an active control and guidance mechanism to steer the system toward potential faults.

Cotroneo et al. [17] extended fault injection methods by applying higher-order Markov models to investigate traditional cloud infrastructures under various scenarios, including systems under workload and idle conditions. While their solution shares many goals and methods with my work, my approach specifically

targets extreme conditions and incorporates a smart steering mechanism tailored for the cloud-edge continuum.

2.3 Query-based and Test Case Generation Methods

Dogga et al. [18] introduced Revelio, an ML-based assistant trained on historical bug reports and system logs. The assistant generates debugging queries by leveraging the basic functionalities of existing tools and can inject faults into distributed systems to actively manipulate their behavior. However, their approach is broader in scope, lacks a formal foundation, and does not provide proven use cases directly related to my work.

Pontillo et al. [19] proposed FERRARI, a failure reproduction tool that generates test cases and analyzes stack traces to identify faults. While their tool efficiently aids in reproducing failures for manual debugging, it does not focus on ML-guided debugging or incorporate a formal modeling background as presented in my work.

2.4 Further Related Works

More related work has been described in our latest related paper [5], this section highlighted additional recent and relevant contributions in a complementary and structured manner.

3 Methodology

3.1 Overview and Problem Statement

The applied procedure consists of several steps: (i) First, simulations are performed based on the system model, and the collected log data is used for machine learning purposes. (ii) Based on the labelled simulation data, a supervised training process is carried out for the ML model. (iii) Once the model has been trained, it is used to guide the traversing process for the debugger, and (iv) I demonstrate how the estimated distance values provided by the model inform the subsequent search directions. (v) Finally, the method is validated to ensure its effectiveness.

The presented procedure is based on the principle that while certain faults are deterministic, their occurrence and manifestation follow a stochastic process with an unknown probability distribution. Furthermore, it is assumed that a fault can be repeatedly reproduced by traversing the same nodes in the state space based on replay techniques (see examples in [8] [9]). However, the same fault can also be triggered through alternative means, i.e. they might be reached in various paths during actively controlled debugging sessions.

3.2 The Base-Line Model

To provide clear understanding, I have selected a basic example to demonstrate the procedure as follows.

The Producer-Buffer-Consumer primitive [5] in cloud continuum blueprints (elementary part of load balancers and temporary data storage) was analyzed for debugging purposes with the PRISM modelling and simulation tool [6] described as a discrete-time (t = 1, 2, ..., T) stochastic process, where the state (Producer, Buffer, Consumer) changes in unpredictable manner over time.

Let $S_t = (P_t, B_t, C_t)$ represent the state space at time t, where:

P_t: the state of the Producer at the t-th time step,

B_t: the state of the Buffer at the t-th time step (number of elements),

C_t: the state of the Consumer at the t-th time step.

The state space (S_t) thus represents the actual state of the Producer-Buffer-Consumer system, where:

$$S_t = (P_t, B_t, C_t), \quad P_t, C_t \in \mathbb{N}, \quad B_t \in \{0,1, \dots, K\}. \tag{1}$$

The state transition probabilities, which describe the system's evolution, are determined by stochastic events:

$$P(S_{t+1} \mid S_t): \text{ the transition probability from state } S_t \to S_{t+1} \tag{2}$$

The state transitions ($S_t \to S_{t+1}$) meet the following rules:

$$\text{If } P_t = 1, \text{ then } B_{t+1} = B_t + 1, \text{ if } B_t < K. \tag{3}$$

$$\text{If } C_t = 1, \text{ then } B_{t+1} = B_t - 1, \text{ if } B_t > 0. \tag{4}$$

The activation of P_t, B_t, and C_t occurs randomly, with probabilities following a uniform distribution.

The Buffer has a limited capacity, defined as

$$B_t \in \{ 0,1,2, \dots, K\}. \tag{5}$$

According to our previous studies [5], a class of faults can be efficiently captured by applying autoencoders, particularly when a buffer reaches or exceeds its storage capacity or becomes empty. Such potentially erroneous situations can be labeled (or registered) using autoencoders in complex cases (even without any knowledge on the topology), or this simple rule can be applied if the relevant system configuration details (including topology) are known:

$$F_t = \begin{cases} 1 & \text{if } B_t \geq K \text{ or if } B_t < 0 \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

Based on the fault occurrence, we define a derived variable D_t as Distance to Fault. At each time step t, the distance to the fault can be computed as follows:

If $F_t = 1$, then $D_t = 0$ (a fault has occurred).

Otherwise, D_t represents the distance to the fault in terms of time steps. (7)

Since the states of the Producer-Buffer-Consumer primitive change in discrete time and transitions occur probabilistically, the process can be modeled as a discrete-time Markov chain. In this model, the state of the system at time t, depends only on the previous state $t-1$, i.e.,

$$\boldsymbol{P}(S_{t+1} \mid S_t, S_{t-1}, \dots, S_0) = \boldsymbol{P}(S_{t+1} \mid S_t). \tag{8}$$

The state of the system is determined by the states of the Producer, Buffer, and Consumer. The state transition probabilities are governed by the transition matrix $\boldsymbol{P}(S_{t+1} \mid S_t)$, which specifies the likelihood of moving from one state to another. The state transitions follow these rules:

If the Producer is activated: $P_{t+1} = P_t + 1$ (9)

If the Buffer is activated:
$$\begin{aligned} B_{t+1} &= B_t + 1 \\ P_{t+1} &= P_t - 1 \end{aligned} \tag{10}$$

If the Consumer is activated and $B_t > 0$,

then the new state is:
$$\begin{aligned} B_{t+1} &= B_t - 1 \\ C_{t+1} &= C_t + 1 \end{aligned} \tag{11}$$

Therefore, in this model, state transitions occur probabilistically, and the extreme condition (with potential fault) arises when the buffer reaches or exceeds its maximum capacity K or becomes empty. The derived variable D_t measures proximity to the potentially fault state, allowing an analysis of how close the system is to failure at any given time step.

The goal of the simulation is data generation, which serves as a data collection process describing the dynamics of the Producer-Buffer-Consumer primitive. During the simulation phase, the primary focus is to gather data on the system's behavior. Throughout the simulation, the following dataset is collected:

$$(S_t, D_t) \tag{12}$$

where $S_t = (P_t, B_t, C_t)$ represents the system state, and D_t is the distance to the potential fault.

Using the trained model, I employ the data generated by the simulation (or data derived from it) to train a neural network in a supervised manner. The task of the neural network is to estimate the distance to the possible fault state for each system state.

The decision making rule, which determines how the state space should be explored, and which directions are worth pursuing to move closer to potential faults, works as follows.

With the help of the trained model, in any given state (selected as the starting point), the distance to potential faults is estimated for all possible next states. Based on these estimates, the next state is chosen such that it minimizes the distance to a suspicious state to be debugged. This approach enables the exploration of the state space in the direction most likely to encounter a fault.

3.3 Conceptions for ML Model Training

Let us assume a system with a state space so large that it cannot be fully explored, but certain parts of it can still be observed. From these observations, a dataset is generated, which includes the system states and a corresponding target variable.

The target variable represents the distance to a potential fault state and is measured as follows: for each observed state, the target variable quantifies how far the state is from a potential fault, providing a meaningful numerical measure of proximity to failure. This distance serves as the basis for further analysis and exploration of the state space.

3.4 ML Model Training to Estimate Distance to Faults

Based on the collected data, the next step was formulated as a supervised learning task. The objective of this task is to estimate the distance to the suspicious state (i.e., the number of steps remaining until a fault may occur in extreme conditions) based on the system's states. To achieve this, the data generated during the simulation was transformed into training data.

Specifically, during the simulation, I conducted 300 independent runs, where each run consisted of 70 steps. At every step, the current state (P_t, B_t, C_t) was recorded, along with a value Δ_t, which represents the temporal distance to the potential fault state. Thus, the training dataset was structured such that each row was represented in the form

$$(X_t, \Delta_t) \tag{13}$$

where:

$X_t = (P_t, B_t, C_t)$, represents the system state at time t,

$\Delta_t \in \mathbb{N}$, represents the number of steps remaining until the fault occurs, if it occurred at all.

Therefore, this structure enabled the development of a supervised learning model that uses system states to predict the temporal proximity to potential fault states.

The complete training dataset consisted of 300 × 70 = 21,000 samples taking into consideration the size of the given state space and some preliminary experimental results. However, cases that occurred after a detected fault in the simulation were excluded. As a result, the final training dataset contained 18,192 observations, each corresponding to a system state and its associated distance to the fault state.

The prepared training data was used to train a Multi-Layer Perceptron (MLP) neural network. The MLP architecture consisted of two hidden fully connected layers, followed by an output layer that produced a scalar estimate for Δ_t. The first hidden layer contained 20 neurons, and the second hidden layer contained 10 neurons leveraging on some preliminary experiments. For training, I applied standard gradient-based optimization (SGD), with the learning process being monitored on a pre-separated validation set to ensure the model's generalization capability. The neural network employed the LeakyReLU activation function (with an alpha value of 0.1) between the MLP layers, allowing it to represent outputs in both linear and nonlinear ranges. At the output layer, a linear activation function was used, as the target variable Δ_t is a continuous value ranging between -1 and 70. This choice ensures that the output can appropriately capture the temporal distance to the potential fault state.

The training process was conducted iteratively using batch size of 512, determined empirically through trial and error. This batch size provided a balance between fast convergence and smooth error reduction. The learning rate was initially set to 0.001, and the model was trained for 100 epochs. After each epoch, the model's performance (error) was evaluated on the validation set, which comprised 20% of the total dataset. The number of epochs was determined based on the principle that training should stop when the error on the validation set begins to increase, indicating potential overfitting. This stopping criterion was also established empirically.

Before training, MinMax normalization was applied to both the input and output data to accelerate the neural network's learning process. Specifically, the original data was transformed into the $X[0,1]$ while preserving the data's monotonicity. The transformation was performed as follows:

$$x' = \frac{x - x_{\min}}{x_{\max} - x_{\min}} \tag{14}$$

where x_{min} and x_{max} present the minimum and maximum values of the states S (or the target variable) within the training dataset. This normalization was necessary because the optimization process, which updates the neural network's weights, is more effective when both the input data and the output values are on a similar scale—such as the $[0,1]$ range. By scaling the data to this range, the training process became more efficient and stable, ensuring better convergence of the model.

In the training dataset, the input X consists of states generated by the simulation:

$$X = \{S_t = (P_t, B_t, C_t)\} \tag{15}$$

The output y corresponds to the distance to the potential fault state for each system state:

$$y = \{D_t\} \tag{16}$$

Initially, I employed an MLP model to learn the mapping $X \rightarrow y$. The model's task is to estimate D_t for each state:

$$\widehat{D}_t = \text{model}(S_t) \tag{17}$$

As this is a regression task, the Mean Squared Error (MSE) was used as loss function during training. The MSE is computed as follows:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2, \tag{18}$$

where N is the number of samples, y_i is the true or observed output value, and $\hat{y}_i$ is the predicted output value by the model.

3.5 Steering Method towards Extreme Situations

The steering process can be summarized in 3 steps:

1. Identify Possible Next States S_{t+1}

   Starting from a given state S_t, we determine all possible next states based on the following rules:

   If $P_t > 0$ and $B_t < K$, then $S_{t+1}^{Buffer} = (P_t - 1, B_t + 1, C_t)$ (19)

   If $B_t > 0$, then $S_{t+1}^{Consumer} = (P_t, B_t - 1, C_t + 1)$ (20)

   $S_{t+1}^{Producer} = (P_t + 1, B_t, C_t)$ (21)

   In general cases, all the enabled state transitions must be taken into account as we described in [7] [8] [9].

2. Distance Estimation $\widehat{D}_{t+1}$

   Using the trained model, the estimated distance to the potential fault is calculated for all possible S_{t+1} states:

   $$\widehat{D}_{t+1} = \text{model}(S_{t+1}) \tag{22}$$

3. Decision Rule

   Among all possible S_{t+1}, we select the state that minimizes the estimated distance to the suspicious state:

   $$S_{t+1}^{*} = \underset{S_{t+1}}{\text{argmin}}\widehat{D}_{t+1} \tag{23}$$

   where:

   (P_t, B_t, C_t): current state

   $\text{model}(\cdot)$: trained MLP model that estimates the distance to the fault state

   K: max. capacity of the Buffer, by or beyond which a fault may occur

   S_{t+1}^{*}: the best next state that minimizes the estimated distance to the fault

Algorithm 1

Steering method for state space exploration toward suspicious conditions

Algorithm 1 Steering Based on MLP Model: Selecting the Next State Based on the Estimated Distance to Fault

Require:
(P_t, B_t, C_t): Current state
model$(\cdot)$: Trained MLP model estimating the distance to the fault
K: Buffer's maximum capacity beyond which a fault occurs

Ensure:
S^*_{t+1}: The best next state that minimizes the estimated distance to the fault

procedure SELECTNEXTSTATE(P_t, B_t, C_t, model)
  $\mathcal{S}_{t+1} \leftarrow \emptyset$ ▷ Set of next possible states
  best_state $\leftarrow$ None ▷ Initialize the best next state
  min_distance $\leftarrow \infty$ ▷ Initial minimum distance
  ▷ Step 1. Generate neighboring states
  if $P_t > 0$ **and** $B_t < K$ **then**
    Add $(P_t - 1, B_t + 1, C_t)$ to $\mathcal{S}_{t+1}$
  end if
  if $B_t > 0$ **then**
    Add $(P_t, B_t - 1, C_t + 1)$ to $\mathcal{S}_{t+1}$
  end if
  Add $(P_t + 1, B_t, C_t)$ to $\mathcal{S}_{t+1}$ ▷ Always allow increasing the Producer
  ▷ Step 2. Estimate distance and select the best state
  for $S \in \mathcal{S}_{t+1}$ **do**
    $\hat{D} \leftarrow$ model(S) ▷ Estimate the distance using the model
    if $\hat{D} <$ min_distance **then**
      min_distance $\leftarrow \hat{D}$
      best_state $\leftarrow S$
    end if
  end for
  return best_state
end procedure

Therefore, this approach (see Algorithm 1) systematically explores the state space by evaluating all possible next states and choosing the one most likely to bring the system closer to a potentially fault condition, as predicted by the trained model.

4 Experiments and Measurements

4.1 First Stage of Experiments (Use Cases 1 and 2)

Two use cases are examined based on the previously discussed Producer-Buffer-Consumer primitive:

1. Use Case 1 (P – B – C)

   In this scenario, a single Buffer connects the Producer and the Consumer. The extreme condition (potential fault state) was defined such that a fault occurs if the Buffer's current value drops below 0 or reaches/exceeds 20.

2. Use Case 2: P – (B1, B2) – C

In this scenario, two buffers, B1 and B2, are placed in parallel between the Producer and the Consumer. The Producer can send data to either buffer, and the Consumer can extract data from any buffer that is not empty. The extreme condition was defined similarly to Use Case 1 but applied to both buffers. Specifically, a fault occurs if the value of either buffer falls below 0 or reaches/exceeds 20.

In summary, Use Case 1 examines a single buffer for faults, while Use Case 2 involves two parallel buffers for the same conditions.

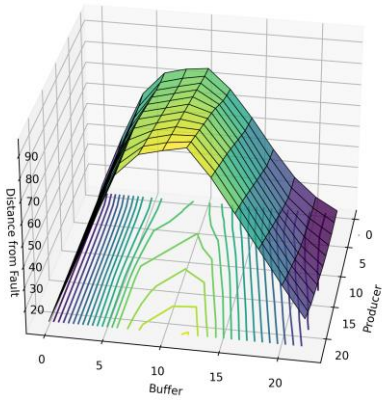

Figure 2
Estimated distances from potential fault for Use Case 1: P-B-C

In this example, the initial state was defined as P=10, B=10, C=10 to ensure clarity and interpretability. The buffer's value was initialized equidistant from both potential fault thresholds, which were defined as occurring when the buffer value falls below 0 or reaches/exceeds 20.

Figure 2 illustrates the surface plot generated by the trained MLP model. It shows the estimated distance to an extreme situation (potential fault state) under different combinations of Buffer (B) and Producer (P) states, with the Consumer (C) state value fixed at 10. The surface reveals how the model predicts the distance to a suspicious situation based on variations in Buffer levels and Producer activity.

One of the key observations related to the behavior of the Buffer: the model predicts that as the buffer value decreases toward 0 or increases toward 20, the estimated distance to the potential fault state also decreases. This aligns with the fault thresholds and reflects the expected system behavior, i.e. the closer the buffer value is to either extreme, the fewer steps are required for a fault to occur.

Another observation shows the independence from the Producer: the model's predictions are unaffected by changes in the producer's value. This demonstrates that the Producer state is irrelevant for estimating the distance to the potential fault state under the given conditions. This behavior matches the expected system

dynamics since the Buffer's state alone determines proximity to a fault in this example.

In summary, the results confirm that the trained MLP model is able to predict the distance to a potential fault. It captures the key relationships between the buffer state and fault proximity while correctly identifying that the producer's state does not influence the outcome.

Table 1
Steering in action toward a potential fault in Use Case 1: P-B-C

Step	**P**	**B**	**C**	**Distance**	**P**	**B**	**C**	**Distance**	**P**	**B**	**C**	**Distance**
0	**10**	**10**	**10**	85.29	**3**	**15**	**10**	32.02	**3**	**7**	**10**	77.35
1	9	11	10	79.09	2	16	10	27.10	3	6	11	69.00
2	8	12	10	70.68	1	17	10	22.18	3	5	12	57.80
3	7	13	10	62.22	0	18	10	17.25	3	4	13	46.76
4	6	14	10	53.76	1	18	10	16.52	3	3	14	35.63
5	5	15	10	45.30	0	19	10	11.60	3	2	15	24.24
6	4	16	10	36.84	1	19	10	10.87	3	1	16	12.91
7	3	17	10	28.38	0	20	10	5.94	3	***0***	17	1.58
8	2	18	10	19.92	1	***20***	10	5.22				
9	1	19	10	11.46								
10	0	***20***	10	5.00								

If we want to analyze, based on the trained model, the direction we should take to reach a potential fault starting from any arbitrary initial state, the process can be carried out as follows. In this example, three test cases with different initial states (see Step 0 in Table 1) are examined.

Table 1 shows the actual steering paths of states as determined by the algorithm. It is presented top to bottom, where each row represents a consecutive state, see Columns P, B, and C. For each state, the trained model calculates the estimated Distance to the potential fault for all possible neighboring states. The table only displays the selected states, i.e. the best paths (shortest route to decrease the distance from the potential fault state).

In Use Case 2, the primitive has been extended to provide a higher level of availability or throughput: one Producer (P); two Buffers (Buffer1, Buffer2) for storing elements independently and in parallel; one Consumer (C).

The results on Figure 3 and Table 2 demonstrate that, starting from any arbitrary initial state, the steering process consistently directs the state space traversal toward the nearest extreme state based on the model's predictions. In this example, a potential fault occurs if any of the following conditions are met: Buffer1 drops below 0 or reaches/exceeds 20, or Buffer2 drops below 0 or reaches/exceeds 20.

When both buffers have values of 10, the system is at its maximum distance from a potential fault state, representing a point of equilibrium. However, moving in any direction along the buffer values brings the system closer to an extreme condition at an accelerating rate.

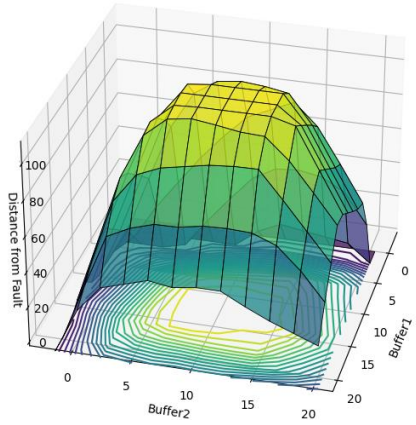

Figure 3
Estimated distances from potential fault for Use Case 2: P-(B1-B2)-C

An important observation is that estimated fault distances by the model do not need to be perfectly accurate for the steering algorithm to be effective. The model remains capable of determining the correct direction to move within the state space, minimizing the distance to a potential fault state with each step. This highlights the robustness of the trained model: even if the distance predictions are slightly off, the model successfully identifies the best path to approach extreme conditions.

Table 2
Steering in action toward a potential fault in Use Case 2: P-(B1-B2)-C

Step	**P**	**B1**	**B2**	**C**	**Distance**	**P**	**B1**	**B2**	**C**	**Distance**
0	**10**	**11**	**10**	**10**	86.47	**5**	**6**	**10**	**10**	83.03
1	9	12	10	10	85.48	5	5	10	11	72.34
2	8	13	10	10	84.48	5	4	10	12	61.65
3	7	14	10	10	72.95	5	3	10	13	50.96
4	6	15	10	10	59.38	5	2	10	14	40.27
5	5	16	10	10	45.81	5	1	10	15	29.57
6	4	17	10	10	32.24	5	***0***	10	16	9.75
7	3	18	10	10	18.67					
8	2	19	10	10	5.10					
9	1	***20***	10	10	-8.47					

The interpretation of the distance to a potential fault in this simulation requires careful consideration due to the system's probabilistic nature. For instance, in Use Case 2, where two buffers operate in parallel, the results indicate that the estimated distance to an extreme condition for Buffer1 can be approximately 5.1 or 29.57 steps, even when it appears that the system is deterministically just one step away from an extreme condition. This phenomenon arises from the probabilistic nature of the system. Since Buffer1 has relatively low chance of being selected in each step, and the Consumer's actions may further influence the Buffer's state, the real distance to the potential fault increases.

In summary, the results confirm that the trained model successfully steers the system toward potential fault states by predicting distances with sufficient accuracy to guide the search. While the fault distance estimates may not always align perfectly with deterministic expectations, they reflect the probabilistic behavior of the system dynamics. This ensures that the model remains reliable in identifying directions in the state space that bring the system closer to extreme conditions.

In order to examine the effectiveness of the elaborated steering method, test scenarios have been designed and examined with Use Case 2 (with half size buffers). The simulation space was defined such that components P, B1, B2, and C could take arbitrary integer values between 1 and 9 as initial state. (Concerning B1 and B2, the values of 0 and 10 were excluded because the extreme condition would have been immediately detected under those circumstances.) The measurement involved 300 test cases, where the mean number of steps to reach an extreme condition was 3.937, with a standard deviation of 2.361 (see Figure 4).

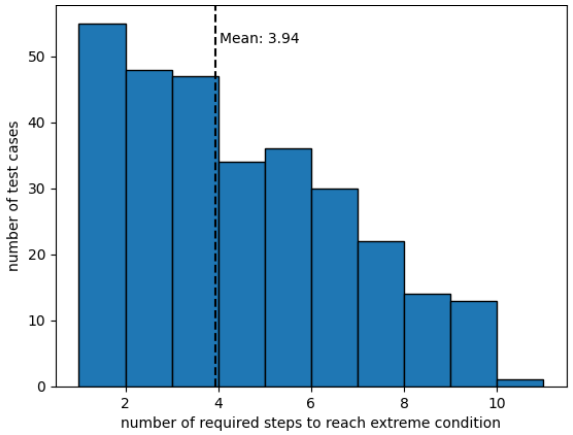

Figure 4
Histogram of required steps toward potential faults based on 300 test cases

4.2 Experiments with Multi-Tier Load Balancer Topology

In order to evaluate our approach and efficiency of the steering process in more complex use cases, I conducted experiments using a multi-tier load balancer topology (see Figure 5), where the load balancers (LB1, LB2, and LB3) can buffer only a limited number of items (B1, B2, and B3). This use case was studied in our previous work [5] using LSTM model without significant results.

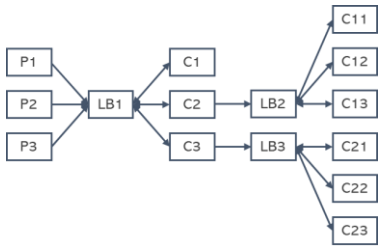

Figure 5
Topology of Use Case 3 with multi-tier load balancers

For this Use Case 3, a more complex 7-layer Dense Neural Network (DNN) architecture was applied after some preliminary experiments (instead of the 2-layer Fully Connected MLP). Each layer contained 10 neurons with SELU activation functions, and the network had a single output node at the end, estimating the distance from the error. In Use Case 3, there were three error conditions: B1, B2, or B3 > 10.

Considering from a steering perspective, only the simulation cases that resulted in errors are highly relevant. Consequently, I performed the training using only the data derived from these error-prone simulation runs. Considering the complexity and state space of the analyzed topology, the number of error cases used for training was based on 3,559 simulation runs that resulted in some form of error. This formed the training dataset, which contained a total of 486,408 observed states.

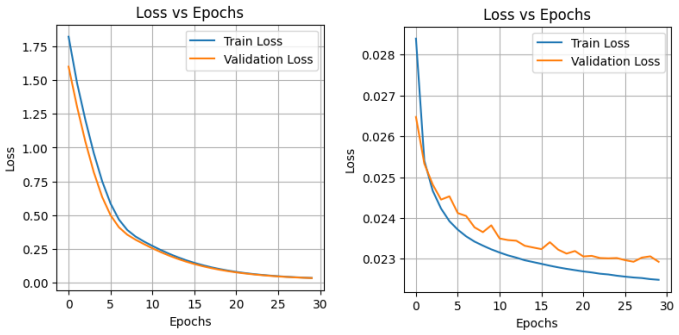

Figure 6
Learning curves (Use Case 3)

The training spanned 5×30 epochs on the same neural network, with an increasing learning rate (0.0001, 0.001, 0.002, 0.005, 0.01) and a decreasing batch size (3200, 800, 320, 80, 32).

Figure 6 illustrates the learning curves for the first and second rounds of training. On the left, the results correspond to a learning rate of 0.0001 and a batch size of 3200, while on the right, they correspond to a learning rate of 0.001 and a batch size of 800.

By the end of the training, the results for the actual and predicted distance values can be seen in Figure 7.

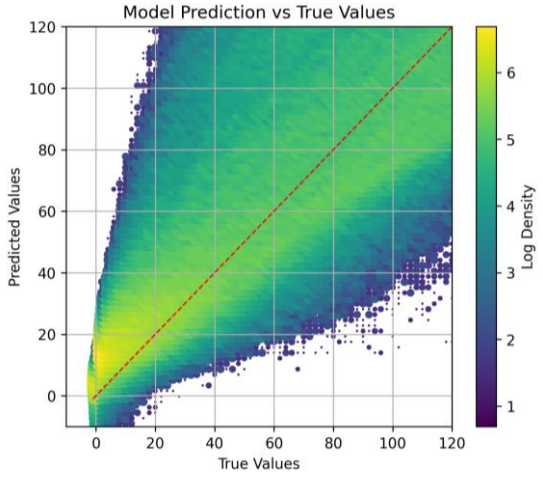

Figure 7
Accuracy of estimation (Use Case 3)

The model becomes increasingly accurate as the system approaches the error state (value of 0). In contrast, the model is less accurate in states farther from the error reflecting the fact that in states farther from the error, there are more possible routes to avoid the error or transition to states where no error occurs. The metrics describing the prediction accuracy were as follows: the Mean Absolute Error (MAE) was 23.102, the Root Mean Square Error (RMSE) was 29.660, the Mean Absolute Percentage Error (MAPE) was 0.71%, the R^2 value was 0.628, and Pearson's Correlation Coefficient was 0.793.

The validation of the trained model on unseen data was performed. For testing the steering algorithm, I conducted a series of 100 measurements, where the initial state represented an operationally observable but previously unseen valid starting condition that had not yet resulted in an error. In this dataset, the distance to the error varied from relatively large to minimal values, and the initial states were selected following a uniform random distribution. Starting from these initial states, the model had to steer the traversing toward a potential fault state, within 20 steps available. If the model managed to steer to the error state within 20 steps, the

steering process concluded with a "Successful" outcome. Otherwise, it resulted in a "Failed" outcome.

In this test, the model successfully steered the traversing to an error in 76 cases and failed in 24 cases. Among the cases where it found the error, the model required an average of 4.72 steps for LB1, 9.5 steps for LB2, and 9.875 steps for LB3 to exceed a value of 10 (i.e., for the internal buffers to saturate). I also conducted additional tests to examine how the steering success rate would change if the model were given 40 or 50 possible steps instead of 20. However, no significant difference was observed in the results. This demonstrates that the model consistently progressed toward the potential fault state along the shortest path available to it.

To illustrate how the steering process actually unfolded with the trained model, I present a specific example in Table 3.

Table 3
Steering in action toward a potential fault in Use Case 3

Step	**P1**	**P2**	**P3**	**B1 (LB1)**	**B2 (LB2)**	**B3*↓ (LB3)**	**C1**	**C2**	**C3**	**C11**	**...**	**C23**	**Dist.**
0*	0	3	1	3	2	2	1	1	2	0		5	
1	0	3	1	2	2	2	1	1	3	0		5	164.0
2	0	3	1	2	2	3	1	1	2	0		5	158.4
3	0	3	1	2	2	4	1	1	1	0		5	142.8
4	0	3	1	2	2	5	1	1	0	0		5	117.7
5	0	3	1	1	2	5	1	1	1	0		5	103.1
6	0	3	1	1	2	6	1	1	0	0		5	74.9
7	0	3	1	0	2	6	1	1	1	0		5	60.3
8	0	3	1	0	2	7	1	1	0	0		5	35.5
9	0	2	1	1	2	7	1	1	0	0		5	30.0
10	0	2	1	0	2	7	1	1	1	0		5	25.6
11	0	2	1	0	2	8	1	1	0	0		5	19.5
12	0	1	1	1	2	8	1	1	0	0		5	17.0
13	0	1	1	0	2	8	1	1	1	0		5	15.8
14	0	1	1	0	2	9	1	1	0	0		5	10.7
15	0	0	1	1	2	9	1	1	0	0		5	9.7
16	0	0	1	0	2	9	1	1	1	0		5	5.08
17	0	0	1	0	2	***10***	1	1	0	0		5	3.07

I highlighted some cells to indicate which two (coupled) states changed in a given step (i.e., which states had a transaction between them compared to the previous step). The table does not include C12, C13, C21, and C22, as their values remained constant throughout the steering process.

From the table, the following observations can be made:

- Until the 4th step, the process was relatively straightforward, as LB3 only needed to take elements from C3.
- However, once C3 was emptied, a different mechanism (pattern) was triggered: C3 first took an item from LB1, and in the next step, LB3 received an item from C3. This pattern repeated as long as there were elements in LB1.
- When LB1 was also emptied in the 10th step, the steering process adopted a third repetitive pattern: LB1 took an element from P2, then C3 took an element from LB1, and finally, LB3 took an element from C3. This sequence is repeated until LB3 reached a value of 10, i.e. an extreme condition.

5 Limitations and Future Work

Due to the nature and complexity of the presented work, one of the limitations is related to the modeling errors: (i) If the simulation fails to sufficiently cover the real state space, the trained model may not generalize well to unseen states. This could lead to unreliable predictions when the system encounters conditions not represented in the training data. (ii) The model may inaccurately estimate distances to potential fault states if the training data is not representative of the full range of system behaviors. In such cases, the model might overfit to certain patterns in the training data, i.e. reducing its ability to perform well on diverse or unexpected inputs.

Addressing these limitations requires careful design of the formal model-based simulation to ensure comprehensive coverage of the state space and representative training data, as well as rigorous validation of the model's predictions.

Future extensions of this approach could involve representing the system states and their transitions to subsequent states using an adjacency matrix. In this context, the adjacency matrix would describe the relation of interacting components. This representation captures all possible relationships between participants and can be used to construct multidimensional time series.

Such a structured representation would provide a more informative input for learning the dynamics of state transitions. For instance, relationships between states could be effectively captured and learned using models like Convolutional Neural Networks (CNNs), which are well-suited for recognizing spatial and temporal patterns in structured data. This enhancement could improve the accuracy and efficiency of the model when predicting distances to extreme states, especially in systems with complex interdependencies and higher number of primitives what we plan to investigate as part of our future work.

Conclusions

In this work, I presented a novel supervised machine learning-based approach to guide debuggers in navigating large, high-dimensional state spaces toward potential fault states. By leveraging a Multi-Layer Perceptron (MLP) and Dense Neural Network (DNN) neural networks, the method effectively predicts the distance to suspicious states and determines optimal directions for exploration. The results from a cloud continuum primitive, including both single and dual-buffer configurations, demonstrate that the proposed method successfully steers the search process toward extreme conditions. The presented approach and its efficiency in the steering process were evaluated using a more complex use case involving a multi-tier load balancer topology.

The proposed approach offers benefits in addressing the challenges of fault detection and debugging in systems where exhaustive state-space exploration is infeasible. The model systematically estimates fault distances and directs search efforts, reducing time and computational costs in debugging cloud continuum environments.

However, limitations such as incomplete state-space coverage and model generalizability are subjects of further research. Future work will focus on enhancing the state representation using (among others) adjacency matrices and exploring graph-based learning techniques and CNNs. These advancements are expected to improve model accuracy and scalability, enabling more robust steering towards extreme circumstances in complex, real-world systems [20].

By integrating machine learning-driven exploration with state-space steering, this work lays the foundation for more intelligent, proactive fault detection techniques. It also may contribute to greater system reliability and performance under extreme conditions.

Acknowledgements

The research was partially supported by the Ministry of Innovation and Technology NRDI Office within the framework of the Autonomous Systems National Laboratory Program. Project no. TKP2021-NVA-01 has been implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme. This work was partially funded by the European Commission's Swarmchestrate Horizon Europe project (No. 101135012). I also thank István Pintye for his valuable advice.

References

[1] Papazoglou, M. P., & Heuvel, W. -J. v. d. (2021) Blueprinting the Cloud. *IEEE Internet Computing*, 15(6), 74-79

[2] Pydi, H., & Iyer, G. N. (2020) Analytical Review and Study on Load Balancing in Edge Computing Platform. *Fourth International Conference on Computing Methodologies and Communication (ICCMC)*, 180-187

[3] Ullah, A., Kiss, T., Kovács, J. et al. (2023) Orchestration in the Cloud-to-Things compute continuum: taxonomy, survey and future directions. *Journal of Cloud Computing*, 12, 135

[4] Beszédes, Á. (2019) Interdisciplinary Survey of Fault Localization Techniques to Aid Software Engineering. *Acta Polytechnica Hungarica*, 16 (3) 207-226

[5] Lovas, R., Rigó, E., Unyi, D., & Gyires-Tóth, B. (2023) Experiences With Deep Learning Enhanced Steering Mechanisms for Debugging of Fundamental Cloud Services. *IEEE Access*, 11, 26403-26418

[6] Kwiatkowska, M., Norman, G. & Parker, D. (2011) PRISM 4.0: Verification of Probabilistic Real-Time Systems. *Lecture Notes in Computer Science*, 6806, 585-591

[7] Lovas, R. & Kacsuk, P. (2007) Correctness debugging of message passing programs using model verification techniques. *Proceedings of the 14th European conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface (PVM/MPI'07)* 335-343

[8] Ligetfalvi, B., Emődi, M., Kovács, J., & Lovas, R. (2021) Fundamentals of a Novel Debugging Mechanism for Orchestrated Cloud Infrastructures with Macrosteps and Active Control. *Electronics*, *10*(24), 3108

[9] Kovács, J., Ligetfalvi, B., & Lovas, R. (2024) Automated Debugging Mechanisms for Orchestrated Cloud Infrastructures With Active Control and Global Evaluation. *IEEE Access*, 12, 143193-143214

[10] Gan, Y., Liang, M., Dev, S., Lo, D., & Delimitrou, C. (2022) Practical and Scalable ML-Driven Cloud Performance Debugging With Sage, *IEEE Micro*, 42(4), 27-36

[11] Huang, J., Jiang, Z., Liu, J., et al. (2024) *Demystifying and Extracting Fault-Indicating Information from Logs for Failure Diagnosis*. IEEE 35th International Symposium on Software Reliability Engineering (ISSRE), 511-522

[12] Shahane, V. (2022) Investigating the Efficacy of Machine Learning Models for Automated Failure Detection and Root Cause Analysis in Cloud Service Infrastructure. *African Journal of Artificial Intelligence and Sustainable Development*, 2(2), 26-51

[13] Yu, X., Ye, K., He, D., et al. (2024) MLPing: Real-Time Proactive Fault Detection and Alarm for Large-Scale Distributed IDC Network. *IEEE 44th International Conference on Distributed Computing Systems (ICDCS)*, 913-924

[14] Han, X., Pasquier, T., Ranjan, T., & Goldstein, M. (2017) FRAPpuccino: Fault-detection through Runtime Analysis of Provenance. *Proceedings of the 9th USENIX Conference on Hot Topics in Cloud Computing (HotCloud'17)*, 18

[15] Kumar, T., Yashika, Singhal, A., Yashvardhan, & Priyadarshini, R. (2024) Early System Failure Detection through System Log Analysis: An LSTM Approach. (2024) *15th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, 1-7

[16] Cao, Q., & Niu, H. (2022) Higher-order Markov Graph-based Bug Detection in Cloud-based Deployments. *IEEE International Performance, Computing, and Communications Conference (IPCCC)*, 153-160

[17] Cotroneo, D., De Simone, L., & Liguori, P. (2020) Fault Injection Analytics: A Novel Approach to Discover Failure Modes in Cloud-Computing Systems. *IEEE Transactions on Dependable Systems*. 19(3), 1476-1491

[18] Dogga, P., Narasimhan, K., Sivaraman, A., Saini, S., Varghese, G. & Netravali, R. (2022) Revelio: ML-Generated Debugging Queries for Finding Root Causes in Distributed Systems. *Proceedings of Machine Learning and Systems*, 4, 601-622

[19] Pontillo, V., Vandercammen, M., & Verbelen, S. (2024) FERRARI: FailurE Reproduction through automatic test cAse generation and stack tRace analysIs. *BENEVOL24: The 23rd Belgium-Netherlands Software Evolution Workshop, 177-189*

[20] Nagy, E., Hajnal, Á., Pintye, I., Kacsuk, P. (2019) Automatic, cloud-independent, scalable Spark cluster deployment in cloud. *CIVIL-COMP PROCEEDINGS*, 112, 26