Received 27 December 2022, accepted 16 January 2023, date of publication 8 February 2023, date of current version 21 March 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3243201

RESEARCH ARTICLE

Experiences With Deep Learning Enhanced Steering Mechanisms for Debugging of Fundamental Cloud Services

RÓBERT LOVAS[1], ERNÖ RIGÓ[1], DÁNIEL UNYI[2], AND BÁLINT GYIRES-TÓTH[2]
[1]Institute for Computer Science and Control (SZTAKI), Eötvös Loránd Research Network (ELKH), 1111 Budapest, Hungary
[2]Department of Telecommunications and Media Informatics, Budapest University of Technology and Economics, 1111 Budapest, Hungary

Corresponding author: Róbert Lovas (robert.lovas@sztaki.hu)

This work was supported in part by the European Union Project [collaboration between Institute for Computer Science and Control (SZTAKI) and Budapest University of Technology and Economics (BME)] within the framework of the Artificial Intelligence National Laboratory under Grant RRF-2.3.1-21-2022-00004; and in part by the National Research, Development and Innovation Office (NKFIH) through Hungarian Scientific Research Fund (OTKA) under GrantK 132838. The work of Róbert Lovas was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences.

ABSTRACT Cloud architecture blueprints or reference architectures allow the reuse of existing knowledge and best practices when creating new cloud native solutions. Therefore, debugging of reference architecture candidates (or their new versions) is an extremely crucial but tedious and time-consuming task due to the deployment of complex services in typical multi-tenant and non-deterministic environments. During the debugging/testing/maintenance scenarios, we might be able to achieve greater levels of test coverage (and eventually improved reliability) by modelling and verifying at least their most fundamental building blocks and their interconnections. The main objective of our work is to integrate stochastic modelling and verification techniques based on deep learning methods into the debugging cycle in order to handle large state spaces more efficiently, i.e. by steering the process of traversing state space towards suspicious situations that may result in potential bugs in the actual system with smart steering during the traversal. For this purpose, our presented and illustrated approach combines (among others) Continuous Time Markov Chain modelling (CTMC) techniques with deep learning methods including autoencoder, Long Short-Term Memory (LSTM) and Graph Neural Network (GNN) models. Our experiences are summarized with widespread cloud design patterns including load balancing and service mesh topologies. According to the results, the debugging cycle can be partly automated through the application of deep learning methods. The autoencoders are able to detect erroneous load balancer behaviors (anomalies) in complex configurations; the LSTMs demonstrate implicitly some random nature of the inspected processes, and GNNs exploit the additional topology-related information in service meshes.

INDEX TERMS Cloud computing, deep learning, software debugging, reference architecture, service mesh, formal verification, Markov chains, autoencoder, long short-term memory, graph neural networks.

I. INTRODUCTION

A. REFERENCE ARCHITECTURES

With the spreading of cloud computing technologies, cloud architects and software developers have been equipped with capabilities (among others) to ingest data from heterogeneous data sources, to perform a vast amount of analytical jobs, and to provide rapid response at the same time, however building reliable solutions that enable such advanced services is still not a straightforward process. Cloud architecture blueprints or reference architectures allow the reuse of existing knowledge and best practices when creating new solutions. There are different definitions proposed, e.g., [2], [3], [4], [5], [6] with the main ideas of (i) promoting re-usability, (ii) incorporating best practices, (iii) using high or low abstraction levels, and (iv) serving certain use cases. However, most existing architecture blueprints are either too abstract; not end-to-end; not vendor agnostic

The associate editor coordinating the review of this manuscript and approving it for publication was Porfirio Tramontana.

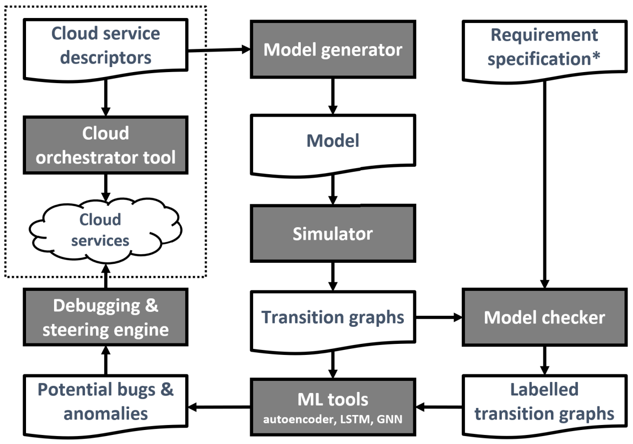

Cloud service descriptors
Model generator
Requirement specification*
Cloud orchestrator tool
Model
Cloud services
Simulator
Debugging & steering engine
Transition graphs
Model checker
Potential bugs & anomalies
ML tools
autoencoder, LSTM, GNN
Labelled transition graphs

FIGURE 1. **Proposed framework for debugging cycle with tools and dataflow.**

or not open-source; or some combination of the above. For example, Microsoft Azure currently offers 769 generic purpose reference architectures.

Moreover, one of the key reasons for applying the reference architecture concept with advanced cloud orchestration tools (e.g. [7]) in practice is to avoid the most common mistakes during the design and software development phases.

B. DEBUGGING CHALLENGES

Debugging reference architecture candidates (or their new versions) is a crucial but tedious and time-consuming task due to the deployment of services in a typically multi-tenant and non-deterministic environment. It poses several new challenges since software designers and testers must face (among others) the probe effect, the irreproducibility, the completeness problem, and also the large state-space that has to be handled somehow during the debugging/troubleshooting phases [8]. By modelling and verifying at least the most fundamental building blocks, we might achieve a higher level of coverage and thus a higher level of reliability, during the debugging, testing and later in the maintenance scenarios.

C. PROPOSED APPROACH

In this paper three widespread and fundamental elements of cloud reference architectures (design patterns) are studied and modelled in detail that provide buffering, load balancing and (micro)service mesh functionalities (see ‘Cloud service descriptors’ in Fig. 1).

The main aim of our work is to improve the developed debugging solution with deep learning methods (see ‘ML tools’ in Fig. 1) to handle the large state space with a higher level of automation, i.e. with smart steering in the traversal of state space towards suspicious situations (potential bugs) using autoencoders, Long Short-Term Memory (LSTM) models, and Graph Neural Networks (GNN).

The automated generation, augmentation and labeling of training data sets (see ‘Labelled transition graphs’ in Fig. 1) as well as the definition of metrics and features for deep learning are crucial in using formal modelling and verification methods, including Continuous Time Markov Chain modelling (CTMC) techniques and the PRISM model-checker [9]. Later, the detected suspicious situations may steer our cloud debugger tool [8] towards the potential faults.

II. RELATED WORKS

As cloud technologies are developing rapidly with more and more applications migrated to cloud environments, stability and management (orchestration) of cloud systems have become crucial. The efficient and continuous monitoring activities are vital to detect faults to accurately operate cloud platforms: cloud monitoring helps to review, observe and manage the complex, orchestrated processes of cloud platforms [10]. However, extended monitoring may create logs with an extremely large number of entries. That is why such logs are difficult to process, and the identification of actual faults on time as well as providing possible preventive actions (before serious errors) are challenging.

The logs produced and collected by a monitoring system could contain memory alerts, unreachable server alerts, application performance alerts, Service-Level Agreement (SLA) expiration and other meaningful information.

The general concept of fault is defined in the manuscript by Farooq et al. [11] as an abnormal condition that may cause a reduction in, or loss of, the capability of a functional unit to perform a required function. Failures are noticeable outcomes of faults and are related to software executions. To prevent failures, the error behind the faults must be discovered before failures occur. Errors are classified as network, software, permanent, transient, intermittent in the survey written by Kumari and Kaur [12]

Cloud monitoring helps to collect continuous data with a large amount of information. To maintain instant and reliable service, fault detection is important. Generally, the size of the failure dataset is relatively small, compared to the overall gathered logs from the cloud routine checks. That creates adversity to extract, evaluate and correct certain faults on the services. Based on the manuscript, written by Swetha and Venkatesh [13], types of faults are classified and related to a particular part of the cloud where erroneous logs are recorded.

An effective method introduced in the paper written by Gao et al. [14] is called the autoencoder model, which is more efficient than traditional methods of fault detection. In this model, unsupervised learning is featured to extract essential characteristic data. A deep autoencoder neural network was introduced to extract erroneous data and provide predictive output as the result of the automatic learning of fault detection in cloud computing.

Fault detection in cloud systems is performed by using the following approaches based on the article by Smara et al. [15]. The first one is called an intrusion or anomaly detection system. The main focus of this type of detection is the network or host intrusions. Anomalies are detected based on behavior analysis. Signature-based and anomaly-based detection are the subgroups of intrusion detection. Signature-based detection contains a predetermined database of already experienced anomalies with particular priority but the anomaly-based detection is looking for atypical patterns inside the log data. An effective anomaly detection system for fault-tolerant network management in cloud data centers is presented particularly in the article by Abbasi et al. [16]. Software-defined networking provides easier administration, network control features, and a programmable console. This feature could be scheduled for the reading network by subnets, and it helps to utilize the best available path in the network and fault management of the cloud network.

In cloud computing, various statistical and ML-based anomaly detection methods are available. The former compares the statistical features of the data with the observed ideal conditions and thus is able to detect unpredictable faults, with limitations. ML-based methods are generally rule-based and machine-learning based. In the rule-based approach, rules about the data and process describing the 'normal' behaviour, are hand-crafted by experts. It is similar to the statistical approach, with more rules and more complex techniques, e.g. fuzzy logic.

In general, error detection can be performed by using machine learning in either a supervised or an unsupervised manner. The ML model learns normal and anomalous behavior from input-output pairs in the case of supervised machine learning. Unsupervised machine learning usually involves learning the normal distribution and detecting errors by measuring the difference between the actual and learned 'normal' behavior. In a sense, this is like statistical and rule-based approaches, but more complex representations of high-dimensional spaces are considered and rules are learned rather than hand-crafted. Combining the different approaches makes them more effective. The use of hybrid methods in cloud computing is, therefore, increasing, which merely requires the expenditure of computational resources.

Other widely used and simple approaches are the heartbeat and pinging methods. In the heartbeat method, a monitoring device continuously checks the fault detector whether a fault occurs. If the detector does not respond before timeout, the heartbeat method considers the device erroneous. The process is reversed in the pinging method. The fault detector sends a message to the monitoring device to confirm the faulty device. Both methods are used for persistent hardware fault detection.

Zhang et al. [17] identified two main categories in cloud fault detection: rule-based detection and detection based on statistics. Rule-based detection methods can be based on simple rulesets that apply to the error message and record components, or basic decision trees can be built using multiple rules and queries. The resemblance of previously found faults are based on other methodologies.

Machine learning methods [17], including Neural Networks (NN) and Support Vector Machines (SVM) are often used as classifiers based on input observations [18], [19], [20], [21].

In tests conducted against topologies, routing, and traffic that are not observed during training, it is demonstrated that GNN models are able to provide accurate estimates of delay and jitter [22]. Also presented in the same paper is the potential of the model for network operation by showing use-cases which demonstrate how it is applied for optimizing delay/jitter routing per source/destination pair and its generalization abilities by reasoning in topologies and routing schemes not encountered during training.

In [23], a problem-specific action space is designed using Deep Reinforcement Learning (Deep RL) agents and GNNs to enable generalization. The proposed GNN-based DRL agent is capable of learning and generalizing over arbitrary network topologies. The DRL+GNN agent was evaluated on 180 and 232 unseen synthetic and real-world network topologies in a routing optimization use case in optical networks, respectively. The results demonstrate that the DRL+GNN agent is capable of outperforming current state-of-the-art techniques in topologies that were not encountered during training.

In [24], an approach is described that enables the prediction of the most significant fault resilience behaviors at web application-level on service mesh, starting from single service to aggregated multi-service management, using model-based reinforcement learning.

Karn et. al. [25] describes an automated testing and resiliency methodology for service mesh based on monitoring traces between microservices, various types of fault injectors (including delay, traffic limiting, and abort), and load testing tool. The faulty microservice link is located through dashboards. Three types of mechanisms are used for resilience or correction of the fault, including scaling, failover, and circuit breaker. This testing and resiliency setup can be used for network troubleshooting and performance measurements for cloud applications.

In [26], the author selected and studied several AI methods for anomaly detection of service mesh-based applications: Support Vector Machine (SVM), Random Forest, Convolutional Neural Network (CNN) and k-means. The results have been integrated into a holistic security and privacy framework.

Wu et. al. [27] proposed a system to help cloud operators to narrow down the potential causes for a given performance issue in microservice architectures. The localized causes are in a fine-granularity, including not only the faulty services but also the culprit metrics that cause the service anomaly. For this purpose, the proposed solution pinpoints a ranked list of potential faulty services by analyzing the service dependencies. Given a faulty service, it applies autoencoder to its relevant performance metrics and leverages the reconstruction errors to rank the metrics. The evaluation showed that their approach can identify the culprit services and metrics with high precision.

Table 1 compares the selected related results according to (i) their supported architecture level, (ii) the selected approach, (iii) the major applied methods, (iv) the main focus

TABLE 1. Comparison of selected related works. (ML: Machine learning).

reference	architecture level	approach	applied methods	focus area	fault injection
Gao [14]	system	ML	autoencoder	generic	manual
Smara [15]	component	self-fault detection	verification (model checker)	component safety	manual
Abbasi [16]	network level	SDN	SLA violation detection	network	manual
P. Zhang [17]	system	ML	support vector machine (SVM)	generic	N/A
Tamura [18]	big data system	ML	clustering (k-means)	SW reliability assessment	N/A
Wang [19]	web apps	ML	feature selection (ReliefF / SVM-RFE)	generic	manual
P. Zhang [20]	system	ML	SVM and decision tree	generic	N/A
X. Zhang [21]	network level	ML	weighted one-class SVM (WOCSVM)	network	manual
Rusek [22]	network level	ML	GNNs	network	N/A
Almasan [23]	network level	ML	GNNs and deep RL	network	N/A
Meng [24]	service mesh	ML/stoch. models	reinforcement learning, CTMC	network	semi-automated
Karn [25]	service mesh	automated testing	rule based error detection	network	semi-automated
Tomas [26]	service mesh	ML	rand. forest, k-means, SVM, CNN	security	manual
Wu [27]	service mesh	ML	autoencoder	SLA	manual
Lovas et. al.	multi-level	ML/stoch. models	autoencoder, LSTM, GNN, MDP/CTMC	generic	automated [8]

area of the research, (v) and the method for fault injection. The literature review leverages partly on [28].

Our approach extends the results of the major related works in several areas including:

- the inspection of cloud architectures at multi-level (from network/component level to services meshes),
- the application of various deep learning methods (autoencoders, LSTMs and GNNs),
- the extensive use of formal modelling e.g. with Markov Decision Process,
- a more generic focus addressing not only the erroneous behaviour of components but the overall reliability (SLA) of the service mesh,
- the active steering of a cloud debugger tool towards the suspicious situations (SLA breach or other failures).

We already proposed a novel cloud debugger method and a framework [8] for this purpose (see 'Debugging & steering engine' in Figure 1) that not only monitors but also actively controls the cloud-based processes leveraging on the so-called macrostep [29] based execution. The approach allows the rule-based evaluation by each macrostep, i.e. by each collective breakpoint set in the execution tree.

III. MODELING: PRIMITIVES OF CLOUD REFERENCE ARCHITECTURES

A. FRAMEWORK

The selection of a suitable modelling framework started with the evaluation of various discrete event simulators. From a broad variety of available tools, the de facto cloud simulation toolkit, CloudSim [30] together with its several derivatives [31], as well as more generic frameworks like JaamSim [32] and SimPy [33] were selected for further review on the basis of available features, recent activity on tool development, availability of documentation and licensing factors.

None of the tools above seemed feasible to describe correlations between intra-application state changes and inter-application communication events in a level of detail that would be sufficient for the targeted cloud service modelling and simulation use cases.

Network event simulation frameworks, like ns-3 [34] and even IP network emulation tools like IMUNES [35] were also considered but we recognized some challenges to describe and simulate stateful cloud services in a sufficient level of detail.

Since the direct support of high level cloud modelling and the simulation of primitives in our specific scenario are hard to manage in a single modelling framework, the modelling requirements were narrowed down to include just a predefined set of application and communication primitives that were considered to be both observable and describable using existing modelling paradigms [36].

A new selection of formal application modelling and verification tools were evaluated based on lessons learned in the course of previous attempts.

Frameworks for modelling parallel software and hardware applications promised a desirable level of abstraction by enabling the description of systems using pseudo code - an approach more friendly to software and systems engineers. For this purpose, TLA+ [37] and its algorithm description language, PlusCal [38] were evaluated in more detail.

Another area of research was focusing on probabilistic model checking applications. From the selection of available frameworks, the PRISM model checker [9] was selected based on its ability to describe, simulate and formally validate models set up in the form of discrete-time Markov chains (DTMCs) or Markov decision processes (MDPs).

Ultimately, the PRISM framework was selected as the project's choice for modelling cloud based services for the purpose of simulation and validation. The choice is based on the simultaneous availability of non-deterministic and probabilistic transitions in MDPs - a feature enabling better description of partially observable error conditions in the context of cloud applications.

The current simulation framework primarily takes advantage of the following PRISM tooling and formal language features:

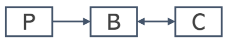


FIGURE 2. Producer (P) and Consumer (C) with Buffer (B).

- Self-supporting *modules* providing means for definition and parallel composition of behavioural patterns in different cloud application modules.
- *Actions* for defining synchronization points between different modules.
- Explicit *labeling* features of interest in a model, providing ease of tracking behaviour changes in a cloud application.
- Cloning of similar model elements (*module renaming*) enabling a lower grade of redundancy in model descriptors.
- *Constants* enabling parametric generation of a large number of modeled application scenarios.
- *Validation* of manually crafted or generated cloud application models.
- Constrained or unconstrained *path simulation* of state transitions in a cloud application model.
- Formal *checking* of models against various properties described as various probabilistic linear temporal logical expressions, such as PCTL [39] - a feature including precise numeric calculation of probabilistic values experienced in application error states.
- *Exporting* model validation, composition, simulation and checking results in various data formats for further processing.

B. BASELINE MODEL: PRODUCER-CONSUMER

As we described in the related works section, one of the most promising and advanced approaches to debugging is when software developers apply active control during the debugging phase, i.e. trying not only to replay but also to enforce different (timing) conditions onto the investigated set of processes (services) in the dynamically changing and non-deterministic environment. The literature [40] illustrated its basic principles and mechanism with the well-known Producer-Consumer problem and its implementation in the P-GRADE programming environment using an inner Buffer process with a circular puffer, message passing paradigm and synchronous communication primitives. The illustration described a typical bug, when the two branches of a conditional (if-then-else) construct have been accidentally swapped and led to an erroneous situation, which was discovered during the macrostep based debugging by traversing and investigating the possible state space of the problem.

In our research this example (see Figure 2) has been taken to build the first PRISM models and perform the first experiments. After defining the constants (such as the buffer size) along with the various probabilities of generating, receiving, serving or consuming an item, the model describes the behaviour of each process with modules: (i) producer, (ii) buffer, and (iii) consumer.

The behaviour of the producer is modelled in the following way:

```
module producer
p: [0..N];
[producing] p>=0 & p<N -> mu:(p'=p+1);
endmodule
```

where 'p' refers to the produced items (maximum is 'N'). According to the sole defined command, this module is able to produce a new item by updating 'p' with 'mu' probability in a synchronized action with the 'buffer' module (labelled as 'producing') once the defined guard is satisfied.

As the next step, we described the buffering mechanism with another PRISM module:

```
module buffer
b: [0..BS];
serving: bool init false;

// normal condition
[producing] b>0 & b<BS & !serving ->epsilon:(b'=b+1);
[con_ready]b>0 & b<BS & !serving ->lambda:(serving'=true);

// buffer is EMPTY
[producing] b=0 & !serving -> epsilon:(b'=b+1);

// buffer is FULL
[con_ready] b=BS & !serving -> lambda:(serving'=true);

// buffer is serving the consumer
[consuming] serving & b>0 -> (serving'=false)&(b'=b-1);

endmodule
```

where 'b' refers to the buffered items (maximum is 'BS'), and 'serving' indicates if the 'buffer' module is occupied with serving the consumer. The internal logic and commands of the 'buffer' module are separated into 4 groups according to the current state of the buffer: (i) normal, i.e. there are some items in the buffer, (ii) buffer is empty, (iii) buffer is full, (iv) serving, i.e. occupied with the consumer.

In normal conditions the 'buffer' module either fetches the item from the producer through the 'producing' synchronized action with epsilon probability and updates 'b', or waits for the consumer to be ready and switches to the serving state with lambda probability. Obliviously, once the buffer is empty or full, only one of these synchronized commands is allowed as the module description shows. During the serving phase, the guard only allows performing the 'consuming' synchronised action with the consumer module that updates 'b' and the serving status as well.

As the last step, the behaviour of the consumer can be described as:

```
module consumer
c: [0..N];
getting: bool init false;
[con_ready] c>=0 & c<N & !getting ->gamma:(getting'=true);
[consuming] getting & c<N->1:(getting'=false)&(c'=c+1);
endmodule
```

where 'c' refers to the consumed items (maximum is 'N'), and 'getting' indicates if the consumer module is occupied

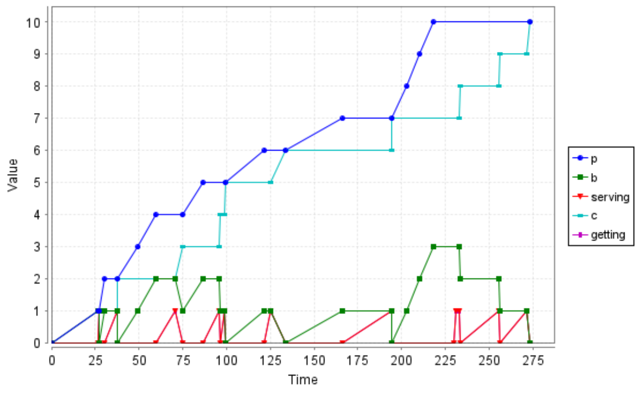
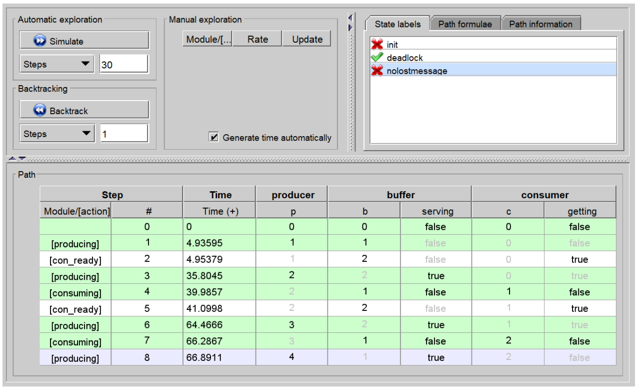


FIGURE 3. Producer-Consumer modelling: typical path of simulation in PRISM.



Steps: 30

Steps: 1

☑ Generate time automatically

✗ init
✓ deadlock
✗ nolostmessage

Step		Time	producer	buffer		consumer	
Module/[action]	#	Time (+)	p	b	serving	c	getting
	0	0	0	0	false	0	false
[producing]	1	4.93595	1	1	false	0	false
[con_ready]	2	4.95379	1	2	false	0	true
[producing]	3	35.8045	2	2	true	0	true
[consuming]	4	39.9857	2	1	false	1	false
[con_ready]	5	41.0998	2	2	false	1	true
[producing]	6	64.4666	3	2	true	1	true
[consuming]	7	66.2867	3	1	false	2	false
[producing]	8	66.8911	4	1	true	2	false

FIGURE 4. Producer-Consumer modelling: typical path of simulation with errors in PRISM.

with getting an item from the ‘buffer’ module. According to the first command, the consumer module notifies the buffer module about its readiness to receive an item with gamma probability, and also updates the ‘getting’ status if the guards allow this synchronized ‘con_ready’ action with the buffer module. In the ‘getting’ phase, the consumer module is able to get the item from the buffer module through the ‘consuming’ synchronized action, which switches back from the ‘getting’ phase and also updates ‘c’.

Figure 3 illustrates a typical path of the simulation where N=10, and BS=3.

The labelling is also supported in PRISM: the ‘nolostmessage’ global property was evaluated by each simulation step successfully in order to verify the most crucial requirement of the model, i.e. the already produced but still unconsumed items are stored in the buffer:

```
label "nolostmessage" = (b=p−c);
```

Similarly to the original work [40], we injected one error manually in the implementation and its model.

```
// normal condition with bug: ***SWAPPING UPDATES***
[producing] b>0 & b<BS & !serving−>lambda:(serving'=true);
[con_ready] b>0 & b<BS & !serving−>epsilon:(b'=b+1);
```

The behaviour of the buffer has become faulty because it switches to the serving state instead of receiving an item in the first command. On the other hand, the buffer module tries to start receiving an item from the producer instead of serving the consumer in the next command.

The simulation of the faulty model and the evaluation of state labels on Figure 4 indicate clearly the states (e.g. in steps #2 and #5) where the ‘nolostmessage’ global property is breached. The simulation and evaluation have even detected later a deadlock situation (step #8).

C. LOAD BALANCING DESIGN PATTERN WITH MULTIPLE SOURCES AND CASCADING

Load balancing is one of the most fundamental elements of cloud-based reference architectures; all cloud computing platforms support this functionality in some way. However,

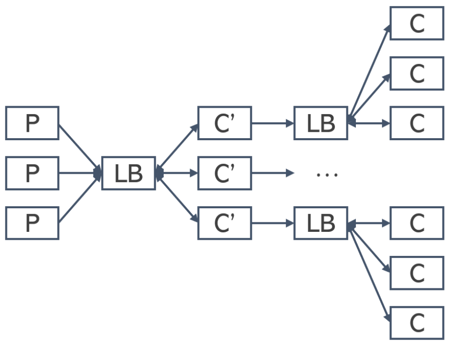


FIGURE 5. Load balancers in a cascaded topology.

load balancing and the unpredictable routing of requests are also a major source of observability difficulties in complex systems. Therefore, modelling and active controlling of load balancers play a crucial role in our research agenda.

Load balancers (LB) might be considered as the extension of the Producer-Consumer problem by adding **multiple consumers** (see boxes labelled by ‘C’ in Fig. 5).

In such generalisation, several load balancing policies can be taken into account. One of the typical policies we followed is when the worker (or consumer) is able to process one request (or a certain number of requests) at once. In this case, the previously introduced model for Producer-Consumer can be extended with multiple consumers. Major changes must be applied particularly in the logic of the ‘buffer’ module, including the multiplication of synchronization actions originally labelled by ‘con_ready’ (see below) and ‘consuming’, and extending the guards to cover the interactions with all consumers.

```
[con_ready1] b>0 & b<BS & !serving1 & !serving2 −> ...
[con_ready2] b>0 & b<BS & !serving1 & !serving2 −> ...
```

In order to simulate cloud systems with a higher level of complexity, **multiple producers** have also been added (cloned) to the original model (see boxes labelled by ‘P’ in Fig. 5) using the module renaming functionality of the PRISM language (see below). The buffer module has also been extended to receive the produced items

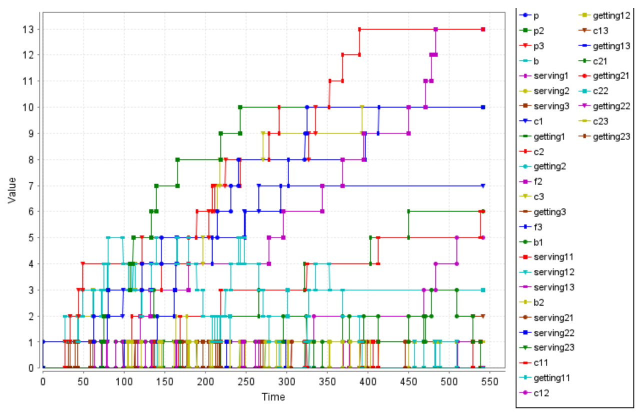


FIGURE 6. **Typical path simulation of a cascaded load balancer.**

with more synchronized actions labelled originally by ‘producing’.

```
module producer2 = producer
[ p=p2, producing=producing2 ]
endmodule

module producer3 = producer
[ p=p3, producing=producing3 ]
endmodule
```

As the next step, our model was elaborated further and extended by **cascading**, i.e. any consumer can be defined as a producer (see boxes labelled by ‘C’ in Fig. 5). In this case, the consumed item is forwarded to another buffer with a pre-defined probability.

This work leveraged the module renaming functionality of the PRISM language to keep the description of the model at a moderate level. As a result, we were able to scale up our modelling capabilities, and observe larger state spaces of more complex systems (see Figure 6).

Similarly to the previous manual error injection, we have swapped two updates, resulting in the breaches of the global predicate at several states (see Figure 7).

The global property can be defined in the following way:

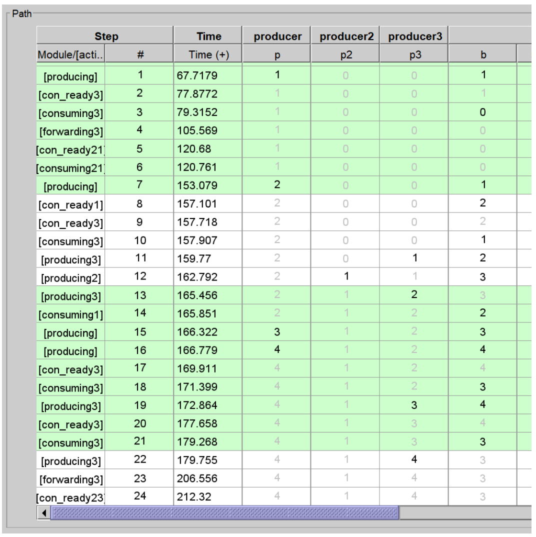
```
label "nolostmessage" = ((b=p+p2+p3–c1–c2–c3) &
(b1=f2–c11–c12–c13) & (b2=f3–c21–c22–c23));
```

where ‘f2’ and ‘f3’ refer to the numbers of forwarded items.

In order to enrich the possible supported use cases, a set of new LB modules have been elaborated by implementing the following strategies: (i) round robin, (ii) random, (iii) least request, (iv) user defined.

For illustration purposes, Figure 8 shows a typical execution of the PRISM model of the ‘least request’ policy with 4 consumers.

We have analysed the complexity of modelling by adding more LBs for a complex service (S[3..6]) and more items to the consumer (Q=10,50,100). The number of states and transitions of the generated state transition graph are summarized in Figure 9.


Path

Step		Time	producer	producer2	producer3	
Module/[acti..	#	Time (+)	p	p2	p3	b
[producing]	1	67.7179	1	0	0	1
[con_ready3]	2	77.8772	1	0	0	1
[consuming3]	3	79.3152	1	0	0	0
[forwarding3]	4	105.569	1	0	0	0
[con_ready21]	5	120.68	1	0	0	0
[consuming21]	6	120.761	1	0	0	0
[producing]	7	153.079	2	0	0	1
[con_ready1]	8	157.101	2	0	0	2
[con_ready3]	9	157.718	2	0	0	2
[consuming3]	10	157.907	2	0	0	1
[producing3]	11	159.77	2	0	1	2
[producing2]	12	162.792	2	1	1	3
[producing3]	13	165.456	2	1	2	3
[consuming1]	14	165.851	2	1	2	2
[producing]	15	166.322	3	1	2	3
[producing]	16	166.779	4	1	2	4
[con_ready3]	17	169.911	4	1	2	4
[consuming3]	18	171.399	4	1	2	3
[producing3]	19	172.864	4	1	3	4
[con_ready3]	20	177.658	4	1	3	4
[consuming3]	21	179.268	4	1	3	3
[producing3]	22	179.755	4	1	4	3
[forwarding3]	23	206.556	4	1	4	3
[con_ready23]	24	212.32	4	1	4	3

FIGURE 7. **Typical path fragment of a cascaded load balancer with errors (only green rows/steps satisfy the ‘nolostmessage’ global property).**

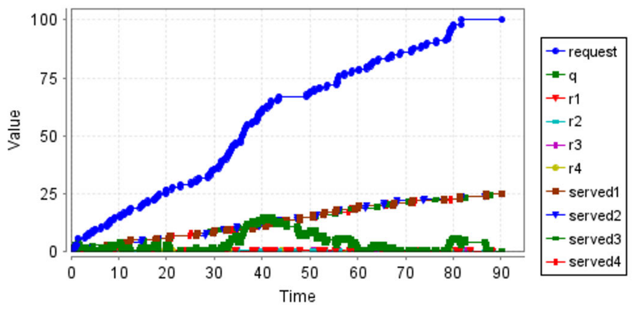
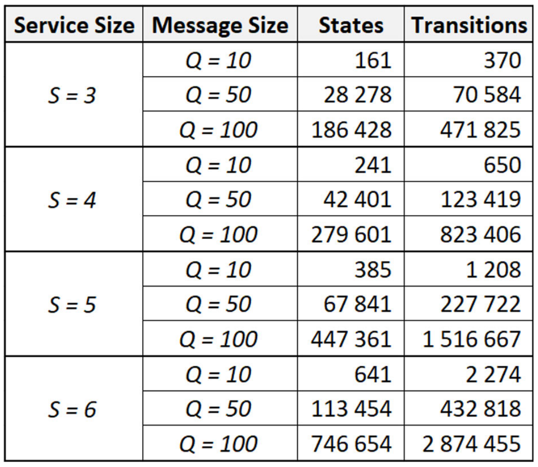


FIGURE 8. **Typical path simulation of a load balancer with ‘least request’ policy.**



Service Size	Message Size	States	Transitions
S = 3	*Q* = 10	161	370
	Q = 50	28 278	70 584
	Q = 100	186 428	471 825
S = 4	*Q* = 10	241	650
	Q = 50	42 401	123 419
	Q = 100	279 601	823 406
S = 5	*Q* = 10	385	1 208
	Q = 50	67 841	227 722
	Q = 100	447 361	1 516 667
S = 6	*Q* = 10	641	2 274
	Q = 50	113 454	432 818
	Q = 100	746 654	2 874 455

FIGURE 9. **Complexity: size of state transition graphs in various LB configurations.**

D. COMPLEX CLOUD ARCHITECTURES (REST-BASED)

In order to generalize our models at a higher level of abstraction, we started to investigate various approaches to describe cloud use cases. As an increasing trend, modern cloud applications often utilize a microservice-based architectural pattern also known as a service mesh [41].

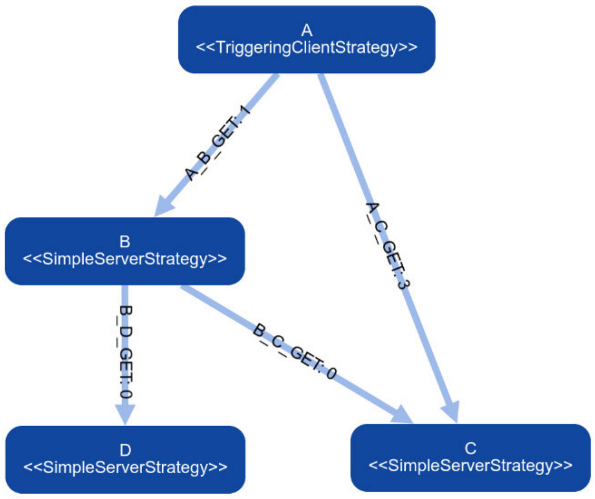


FIGURE 10. Simple service mesh application topology.

In a service mesh, most application functions are implemented as a loosely coupled set of minimalistic web services. The mesh networking component provides various aspects supporting intra-application integration, most prominently the interception and routing of requests between components. In such environments, stateless web service components, utilizing feasible protocols like REST [42] are preferred.

Based on these premises, a model of a simplified service mesh application was devised. The model consists of RESTful service nodes that form a dependency relationship in a form of a directed acyclic graph (DAG). In the model, the root node is considered to trigger requests to its direct descendants, selecting from each with a predefined level of probability. Intermediary nodes are passively listening for requests from their parent nodes; upon receiving a request from a parent node, it is relayed to a selected child node (again, with predefined probability). Finally, the leaf nodes are tasked with answering all incoming requests with either an ‘‘ok’’ (HTTP 2xx) or a ‘‘server failure’’ (HTTP 5xx), with a predefined failure probability rate. Receiving a reply with a failure state triggers a repeated relay mechanism with a predefined number of retries on each intermediary node as well as the root node.

Figure 10 illustrates a simple RESTful service mesh application with a root node (A), an intermediary node (B) and two leaf nodes (C) and (D). Node labels are also representing the strategy of the given node, where ≪TriggeringClientStrategy≫ is the active strategy of the root node, while ≪SimpleServerStrategy≫ is a passive strategy either relaying or directly answering requests. Edge labels are representing REST operations and respective weights for probability calculation.

Based on this approach, a REST modelling framework with a simple domain specific language (DSL) able to define RESTful application models was devised in the Python programming language. The sample application above is defined by the following DSL snippet:

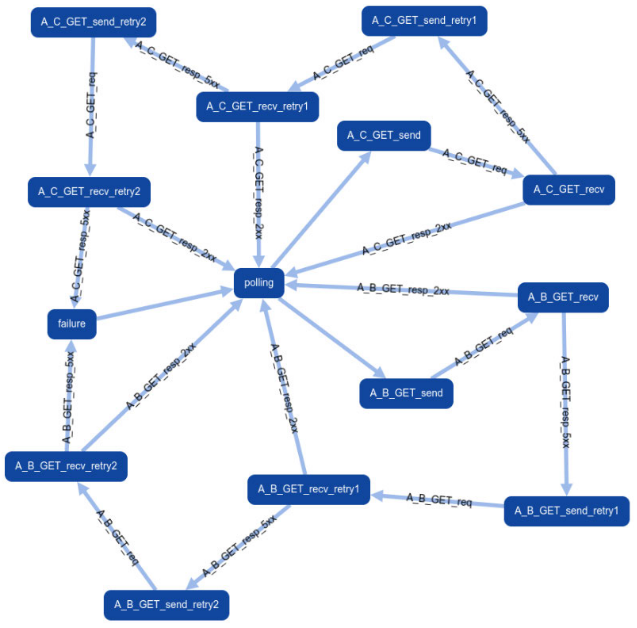


FIGURE 11. Internal state machine representation of node (A).

```
m = RestModel()
a = m.add_node(‘‘A’’, TriggeringClientStrategy)
b = m.add_node(‘‘B’’, SimpleServerStrategy)
c = m.add_node(‘‘C’’, SimpleServerStrategy)
d = m.add_node(‘‘D’’, SimpleServerStrategy)
m.add_flow(a, b, priority=1)
m.add_flow(a, c, priority=3)
m.add_flow(b, c)
m.add_flow(b, d)
```

Based on the DSL above, the modelling framework generates probabilistic state machines for each node, based on its strategy, as well as incoming and outgoing REST flows at that node.

Figure 11 illustrates the internal representation of node (A) with an initial state of ‘‘polling’’, several states representing communication actions between the nodes and a return path to either a ‘‘failure’’ state or directly to the initial state. The ‘‘failure’’ state is visited when the number of retries is exhausted for the node without being able to receive an ‘‘ok’’ (2xx) response. This process is formally described as Algorithm 1.

At this point the DSL model is transcribed to a respective PRISM model by generating module descriptions for all nodes. Following successful validation, RESTful communication flows with arbitrary lengths can be simulated.

The probability of overall application failure can also be expressed by checking the numerical value for the expression $P_{max} = ?[F < NA_{failure}]$, where $A_{failure}$ is the ‘‘failure’’ state of the root node (A) and N is the number of model transitions measured from the initial ‘‘polling’’ state. This property is closely related to application availability for a given time period of N.

Algorithm 1 Algorithm for PRISM Model Generation

```
Input: $N[\{ns, nl, pf, F[\{fl, d, t, p\}]\}]$: set of REST nodes $N$
    with associated strategies $ns$, probability of node failure
    $pf$, labels $nl$ and associated set of flows $F$ with their
    labels $fl$, directions $d$, targets $t$ and probabilities $p$
Output: $M$: set of probabilistic state machine representa-
    tions for all nodes in $N$
    Initialization:
1:  $M \leftarrow [\{empty\}]\ \{\}$
2:  for all $N$ do
3:      if $N[ns] == triggering$ then
4:          for all $F|d = out$ as $F_{out}$ do
5:              $M[N] += req(F_{out}[fl] \rightarrow F_{out}[t, p])$
6:          end for
7:      else if $N[ns] == serving$ then
8:          for all $F|d == in$ as $F_{in}$ do
9:              if $\exists F_{out} : N \in F_{out}[t]$ then
10:                 $M[N] += req(F_{out}[fl] \rightarrow F_{out}[t, p])$
11:                 $M[N] += resp_{2xx}(F_{in}[fl] \leftarrow F_{out}[t, p])$
12:                 $M[N] += resp_{5xx}(F_{in}[fl] \leftarrow F_{out}[t, p])$
13:             else
14:                 $M[N] += resp_{2xx}(F_{in}[fl] \leftarrow N[1 - pf])$
15:                 $M[N] += resp_{5xx}(F_{in}[fl] \leftarrow N[pf])$
16:             end if
17:         end for
18:     end if
19: end for
20: return M
```

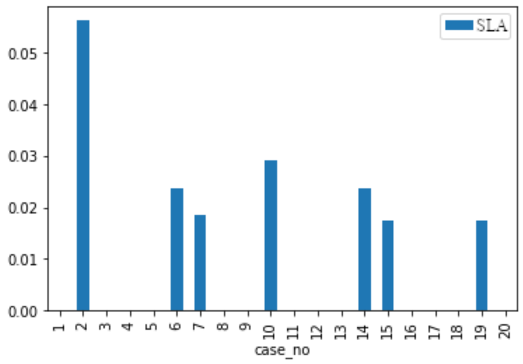


FIGURE 12. Application error probabilities (SLA) for different topologies.

Experimental data involving the generation of cloud application models in the form of random DAG topologies with all other model parameters intact yields significantly different values for P_{max} as described above. Due to its resemblance to application availability, a central component of Service Level Agreements, this quality-like property will be referred to as *SLA* for the purpose of this paper. Figure 12 illustrates *SLA* values for 20 different REST application models (cases) with identical parameters, but different internal topologies.

TABLE 2. Sample model path simulation data (PRISM export file).

step	action	p	b	serving	c	getting
0	-	0	0	false	0	false
1	[producing]	1	1	false	0	false
2	[con_ready]	1	2	false	0	true
3	[producing]	2	2	true	0	true
4	[consuming]	2	1	false	1	false
5	[con_ready]	2	2	false	1	true
6	[producing]	3	2	true	1	true
7	[consuming]	3	1	false	2	false
8	[con_ready]	3	2	false	2	true
9	[producing]	4	2	true	2	true
10	[consuming]	4	1	false	3	false
11	[producing]	5	1	true	3	false

IV. DETECTION OF ERRONEOUS STATES

A. TRAINING DATA GENERATION

Data sets for the first experiments on the baseline buffer producer-consumer model (described in Section III-B) were generated by instrumenting manually crafted PRISM model definition files with substitution points for parameters N (number of items in the buffer), BS (buffer size) and μ (probability of item generation by the producer).

The resulting model templates were instantiated with sweeping ranges $BS[3..24]$, $N[10..80]$ and $\mu[\frac{1}{10}..\frac{1}{4}]$ for both error injected (deadlocked) and error-free models.

For each set of parameters a set of 100 random path simulations were generated and exported in tabular format for the purpose of further analysis. Table 2 illustrates contents of one path simulation data unit with *action* representing synchronization points between modules, columns *p*, *b* and *c* representing internal states of the model as integer values, *serving* and *getting* are values for labels representing logical expressions describing overall module intents.

Based on more complex, load-balanced and cascading producer-consumer models (described in Section III-C), a new collection of training data sets were generated using similar methods as before.

Data sets consisting of 100 simulation paths were generated for load-balanced models with different combinations for 1, 2 and 3 producers and consumers in both error-injected and error-free variations with parameter ranges $N[10..80]$ and $BS[3..40]$.

An addition of data sets with 100 simulation paths were generated for even more complex cascaded models with 3 producers and 3 consumers in both error-injected and error-free variations with parameter ranges $N[5..20]$ and $BS[3..10]$.

At this point, the number of path simulations in the training data sets have reached 13.300 with a total of 1.127.556 events.

It is also important to note that the training data set is purely synthetic (based on model generation and simulation), therefore no additional efforts for reducing noise were necessary. This statement also stands true for the entire scope of the work.

B. AUTOENCODERS

The autoencoder is an artificial neural network in which the output is derived from the input. In general, input and output are the same, and the neural network is subject to some constraints. The input space $\mathcal{X}$ is encoded first into a latent space, $\mathcal{Z}$, by the encoder $E_\phi : \mathcal{X} \to \mathcal{Z}$, where $\mathcal{Z}$ typically meets some constraints (e.g. lower dimension). $\mathcal{Z}$ is then decoded into $\mathcal{X}$ with the decoder $D_\theta : \mathcal{Z} \to \mathcal{X}$, while minimizing the error $L(\theta, \phi) := \mathbb{E}_{x\sim\mu_{ref}}\left[d\left(x, D_\theta\left(E_\phi(x)\right)\right)\right]$, referred to as reconstruction loss. Both $\mathcal{X}$ and $\mathcal{Z}$ are Euclidean space, and ϕ and θ are the trainable parameters of the encoder and decoder, respectively. In case of regression type problems, the reconstruction loss is mostly L2-loss: $L(\theta, \phi) = \frac{1}{N}\sum_{i=1}^{N}\left\|x_i - D_\theta\left(E_\phi\left(x_i\right)\right)\right\|_2^2$. For classification type problems, cross-entropy is applied most: $L(\theta, \phi) = -\frac{1}{N}\sum_{i=1}^{N}\left\|p_i \log D_\theta\left(E_\phi\left(p_i\right)\right)\right\|$, where p_i are one-hot encoded categorical variables. Autoencoders come in many variants, with different input and output data engineering, encoder and decoder structures, and bottleneck constraints. There are a wide variety of applications for autoencoders, including machine translation [43], audio and image generation [44], [45], predictive maintenance and cybersecurity [46], just to name a few.

Autoencoders can be applied effectively in the detection of anomalies (as is done in many predictive maintenance and cybersecurity applications). The occurrence of anomalies is rare, and these phenomena do not form part of the main process. Therefore, in the dataset, anomalies comprise only a small proportion of the data (e.g. less than 1%, typically, otherwise the process would be too malfunctioning). Since neural networks are not effective at learning underrepresented samples from data, these anomalies are very difficult to model, explicitly. Alternatively, when an autoencoder is trained using such data, it learns the distribution of the normal behavior. In other scenarios normal behaviour can be recorded, without any errors. So, in both cases, it is capable of reconstructing the input of data from the output of normal behavior. A well-trained autoencoder will not be able to reconstruct anomalies if the data contains any - therefore, the reconstruction error will be higher.

C. APPLICATION OF AUTOENCODERS FOR ERROR DETECTION

We performed autoencoder-supported error detection in the Producer-Consumer design pattern with gradually increasing complexity: (i) single producer - single consumer, (ii) single producer - three consumers (realising the load balancing design pattern), (iii) three producers - three consumers. In these configurations, producer(s) and consumer(s) were connected by a 'buffer' module. We considered each time step in the simulation as a data point, with features representing the number of items in the individual modules. For instance, in a single producer - three consumers configuration, we can define 5 features: 'p' is the number of items produced by the producer, 'b' is the number of items in the buffer, and 'c1', 'c2', 'c3' are the number of items consumed by each consumer. The number of produced items was set to 10, and the buffer size was set to either 3 or 5.

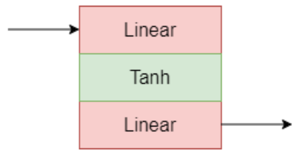


FIGURE 13. Layers of the proposed autoencoder for error state detection.

We also experimented with a cascaded architecture, a hierarchical configuration in which producers and end consumers are connected by three buffers (i.e. the load balancers) and intermediate consumers. Our approach was the same as for the simpler models: each time step in the simulation was considered as a data point, and features represented modules: 3 producers, 3 buffers, 2 intermediate consumers, and 7 end consumers (see Figure 5). Intermediate consumers forward the consumed items with a predefined probability. We also used the number of forwarded items by the 2 intermediate consumers as features, leading to a total of 17 features. The number of produced items was set to either 10 or 20, and the buffer size was set to 10.

In all 4 cases, we generated 200 error-free simulations, and another 200 error-injected simulations (see sections III-B and III-C). We used the 200 error-free simulations to train a simple autoencoder, with the following layers: $\{Linear\} \to \{Tanh\} \to \{Linear\}$ (see Figure 13). For the 3 simpler models, latent space had one fewer dimension than input space, and for the cascaded architecture, we decreased the number of dimensions by 3.

We trained the autoencoder for 100 epochs, with mini-batches of 100 data points. We optimized the network parameters using Adam [47] with a learning rate of 0.001 and mean-squared error (MSE) as loss function. The autoencoder was tested on the 200 error-injected simulations, in which some of the states are erroneous and are guaranteed to run towards deadlock. In the erroneous states, the no-lost message property is not satisfied. For the 3 simpler models, this means $b \neq \sum_i p_i - \sum_i c_i$, and for the cascaded architecture, this means $(b \neq p1 + p2 + p3 - c1 - c2 - c3)|(b1 \neq f2 - c11 - c12 - c13)|(b2 \neq f3 - c21 - c22 - c23)$.

D. RESULTS

In all 4 cases, the autoencoder successfully recognized all the erroneous states by producing significantly larger reconstruction error. Figure 14) is based on a simulation of the three producer - three consumer configuration, but the behaviour is the same for the other 3 configurations we experimented with. The autoencoder trained exclusively on simulations with error-free states.

As it can be seen in Figure 14), when the no-lost message property is not satisfied (denoted by F) during the simılation steps (see the horizontal axis), the loss is orders of magnitude larger than for error-free states (denoted by T). Therefore, by selecting a threshold value (e.g. MSE=0.2), it can be employed as a trustworthy classifier, deciding whether a step in our simulation is loaded with error.

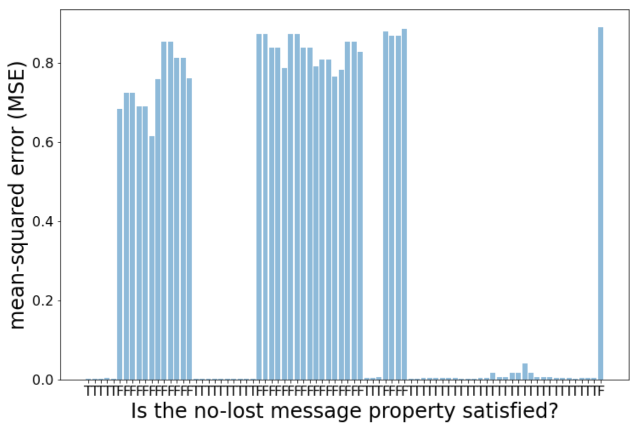


FIGURE 14. Reconstruction loss of the trained autoencoder.

V. STEERING TOWARDS ERRONEOUS STATES

A. GENERATION OF TRAINING DATASET

The experiments in this section are based on the same data sets that are described in Section IV-A.

B. LONG SHORT-TERM MEMORY

Long Short-Term Memory (LSTM) [48] are among the frequently used sequential deep learning models, besides convolutional neural networks [49] and transformers [50]. An LSTM is a recurrent neural network with an internal memory. In the inner memory, there is no operation that can lead to vanishing gradients. The memory is controlled by a gating mechanism, realized as trainable parameters and sigmoid activation functions. The gating mechanism and the inner memory can be described with the following equations:

$$\begin{aligned}
i_t &= \sigma\left(x_t U^i + h_{t-1} W^i\right)\\
f_t &= \sigma\left(x_t U^f + h_{t-1} W^f\right)\\
o_t &= \sigma\left(x_t U^o + h_{t-1} W^o\right)\\
\tilde{C}_t &= \tanh\left(x_t U^g + h_{t-1} W^g\right)\\
C_t &= f_t * C_{t-1} + i_t * \tilde{C}_t\\
h_t &= \tanh\left(C_t\right) * o_t
\end{aligned} \tag{1}$$

where U and W are matrices, and i_t, f_t and o_t are the input matrix, output matrix, input, forget and output gates, correspondingly. C_t represents the inner memory, and h_t is the hidden state. There are a large number of variants of LSTM, including multi-scale approaches [51], [52], [53], [54] and introducing advancements of deep learning [55], [56]. LSTMs have been widely applied in various fields, including speech synthesis [57], financial analysis [58] and weather forecast [59], just to name a few.

C. APPLICATION OF LSTMs FOR ADVANCED STEERING

First, we investigated our baseline model created for the producer-consumer problem with one injected error (see Section III-B) and its execution paths. According to generation of transition graph and simulations with PRISM, the deadlock situations were always predicted in advance by the periodical appearance(s) of the unsatisfied 'nolostmessage' global property when a 'con_ready' action had been completed. Two types of deadlocks can be categorized: Deadlock#1 and Deadlock#2.

In the case of Deadlock#1, there is only one item in the 'buffer' module, and it tries to fetch an item from the 'producer' module via the 'producing' synchronized action. The deadlock will occur because the 'buffer' module is in the 'serving' state but the 'consumer' module is not in the 'getting' state (see Figure 4). In general, the number of 'producing' actions must be greater than the number of 'con_ready' actions by 2 to reach this deadlock situation. The S1 steering rule is straightforward: always prefer 'producing' action instead of 'con_ready', i.e. trying to make the buffer *empty* with more produced items. In other words, always try to avoid the breaching of the global property, since it happens as the result of 'con_ready' action.

In the case of Deadlock#2, there are two items in the 'buffer' module, and the 'consumer' module makes an attempt to notify the 'buffer' module about its readiness to receive an item, while the producer has completed the production of all items (i.e. p=N). The deadlock will occur because the 'buffer' module is not in the 'serving' state but the 'consumer' module is in the 'getting' state. In general, the number of 'producing' actions can not be greater than the number of 'con_ready' actions by 2 to reach this deadlock situation. The SR2 steering rule can be defined as (contrary to SR1) always preferring 'con_ready' action instead of 'producing', i.e. trying to keep the buffer *full* with more consuming. In other words, always try to breach the global property, since it happens as the result of 'con_ready' action.

Since the steering rules SR1 and SR2 are the opposite of each other, the automated generation of such type of steering policies using deep learning methods is challenging. On the other hand, the length of the routes to deadlock situations is quite limited, which may cause further difficulties.

In order to make an attempt to overcome these obstacles, we investigated the models with higher complexity, with more consumers/producers (see Section III-C) and focused on the prediction of the 'nolostmessage' property (instead of the deadlock) with following preliminary results.

We framed steering as a binary classification task, such that our goal is to predict whether the next state is erroneous or not, based on the previous states in the simulation. We used the cascaded architecture with 17 features described in the previous section. The simulations were split into disjoint sequences of length 11, using the first 10 states as a time series in the train set (10-by-17 matrices). We used the 11-th state to determine the label corresponding to each time series, by calculating whether it is erroneous using the no-lost message property:

$(b \neq p1 + p2 + p3 - c1 - c2 - c3)|(b1 \neq f2 - c11 - c12 - c13)|(b2 \neq f3 - c21 - c22 - c23)$

In the final data set, 147 sequences were erroneous and 354 sequences were error-free. We split the data set randomly into train, validation and test sets, with respective lengths 301, 100 and 100. We trained the following neural network on the train set: $\{LSTM\} \rightarrow \{Tanh\} \rightarrow \{Linear\} \rightarrow \{Sigmoid\}$. Different hidden layer sizes (from 2 to 17) and batch sizes were studied. We optimized the network parameters using Adam [47] with a learning rate of 0.001 and binary cross-entropy (BCE) as loss function.

D. RESULTS

Finally, the trained model reached $\approx$ 70% accuracy, which is the proportion of the error-free states, by consistently predicting zeros.

We tried to counterbalance this effect by giving more weight to the negative samples in BCE. Weighting resulted in a few nonzero predictions (some of which were correct), but our model was still unable to learn properly the state changes which led to erroneous states. This can be attributed to the nature of our simulator: among the possible state transitions, it selects the next state by probability, not based on learnable rules.

VI. SERVICE-LEVEL AGREEMENT (SLA) PREDICTION

A. GENERATION OF TRAINING DATASET

In order to improve experiments further towards increasingly realistic situations, training data sets were generated based on the RESTful service mesh modelling approach (described in Section III-D).

Several cases containing simulation data sets were generated for different architecture topologies (based on random DAGs).

With static model parameters of the probability of failure at the leaf nodes ($P_{failure} = 0.1$) and the number of request retries in the case of error ($N_{retries} = 1$) 100 different service mesh models (simulation cases) were generated consisting of a different number of RESTful nodes in the range of [15..30] with a total of 400 service mesh graphs for the whole data set.

For each case a graph description in Graph Exchange XML Format [60] was generated, together with a number of data sets containing path simulations with a length of [10.000..20.000], based on the respective model. A simplified graph description from the generated set is partially presented below for illustration purposes:

```
<gexf version="1.2">
<graph defaultedgetype="directed">
<nodes>
<node id="0" label="0" />
<node id="1" label="1" />
<node id="2" label="2" />
 [... ]
<node id="12" label="12" />
<node id="13" label="13" />
<node id="14" label="14" />
 </nodes>
<edges>
<edge source="0" target="2" id="0" />
<edge source="1" target="8" id="1" />
<edge source="1" target="7" id="2" />
 [... ]
<edge source="12" target="6" id="11" />
<edge source="12" target="11" id="12" />
<edge source="13" target="3" id="13" />
 </edges>
 </graph>
 </gexf>
```

The method for path simulation is the same as described in Section IV-A. The primary difference is that service mesh models are experiencing a cyclic behaviour, therefore no clear termination condition for simulations can be defined. As a result, path simulations with arbitrary lengths can be generated, while the total number of path simulations becomes an irrelevant property. The total number of resulting events in all data sets have reached 19.554.187.

Another distinguished property of the data sets described in this section is the lack of error-injected and error-free variations. As all service models are formally verified to be free of deadlocks, thus another property feasible for classification was to be selected. For this purpose the precise value of SLA (as described in Section III-D) was calculated by formal verification of the generated models and included in each generated service mesh case.

B. GRAPH NEURAL NETWORKS

Graph Neural Networks (GNNs) are frequently used to model real-world networks, defined as a set of interactions between a set of objects. Examples include fake news detection [61], estimated time of arrival prediction [62], and automated drug design [63]. Each data point is a node in the network, and beyond its feature vector, it is also associated with an adjacency vector, representing the connections with the other data points (nodes). Feature vectors are collected into a feature matrix $\mathcal{X} \in \mathbb{R}^{N\times F}$, and adjacency vectors into an adjacency matrix $\mathcal{A} \in \mathbb{R}^{N\times N}$, where F and N denote the number of features and the number of data points, respectively. Since real-world networks are usually very sparse, only a small fraction of $\mathcal{A}$'s entries are non-zero. GNN layers are optimized such that they can effectively manipulate these very sparse matrices [64]. A GNN contains one or more message passing layers, which aggregate information in each of its neighbourhoods:

$$x_i' = \phi\left(x_i, \bigoplus_{j\in\mathcal{N}(i)} \theta\left(x_i, x_j\right)\right)$$

where $\bigoplus$ is a differentiable, permutation invariant function, e.g. sum, mean or max, and ϕ and θ are differentiable functions, e.g. MLPs. In our paper, we used a simple but expressive message passing layer, called the graph

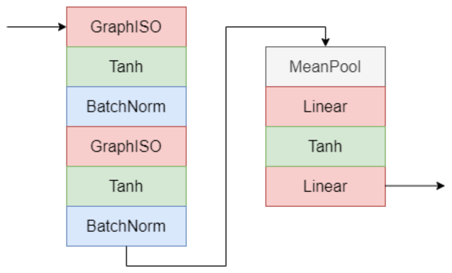


FIGURE 15. **Layers of the proposed neural network for SLA estimation.**

isomorphism layer [65]:

$$x_i' = \phi\left(x_i + \sum_{j\in\mathcal{N}(i)} x_j\right)$$

C. APPLICATION OF GNNs FOR SLA PREDICTION

SLA prediction is a graph-level regression task. We considered nodes of the service meshes as data points, associated with 6 features (number of sent/received GET requests, number of sent/received HTTP 2xx responses, and number of sent/received HTTP 5xx responses). Features from the data set were counted over the whole simulation and divided by the length of the simulation. We split the dataset randomly into train and test set, containing 80% and 20% of the service meshes, respectively. Our goal was to fit a function f such that for any input pair (x_i, a_i), its output $\hat{y}_i$ is as close to the ground-truth SLA value y_i as possible, in terms of squared error $(\hat{y}_i - y_i)^2$.

For this purpose, we proposed a graph neural network with the following layers: $\{GraphIso\} \rightarrow \{Tanh\} \rightarrow \{BatchNorm\} \rightarrow \{GraphIso\} \rightarrow \{Tanh\} \rightarrow \{BatchNorm\} \rightarrow \{MeanPool\} \rightarrow \{Linear\} \rightarrow \{Tanh\} \rightarrow \{Linear\}$, as it can be seen in Fig. 15.

Each hidden layer had dimensionality 8. We trained the neural network for 400 epochs, with mini-batches of 10 service meshes. We optimized the network parameters using Adam [47] with a learning rate of 0.01 and mean-squared error (MSE) as loss function.

D. RESULTS

Our learning curve indicated a successful training process (see Figure 16), and we achieved $4 \cdot 10^{-5}$ loss on both the train and the test set.

We also tracked the absolute error between the output and the ground-truth SLA values during the training, which is a more meaningful metric in terms of interpretability. The mean absolute error of the trained neural network was 0.004 for both the train and the test set (see Figure 17).

We also considered an alternative task in which our goal was to separate service meshes with larger-than-zero SLA from the ones with zero SLA. Using the same graph neural network and training settings, we performed this binary classification task with 100% accuracy. We suspected that perfect accuracy can be bound to a simple property of the

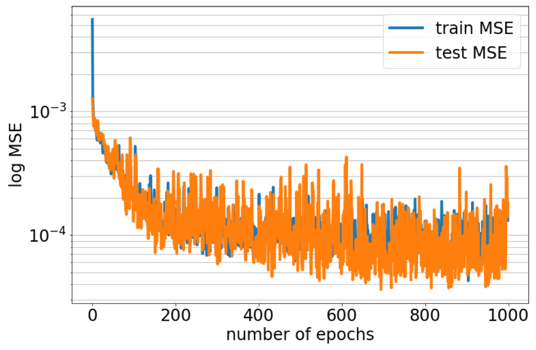


FIGURE 16. **Learning curve for SLA prediction (number of epochs vs. mean-squared error).**

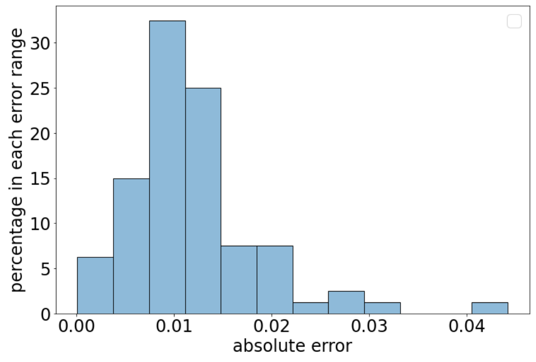


FIGURE 17. **Histogram of absolute errors between predicted and ground-truth SLA values.**

generated graphs. Since faults are generated by leaf nodes, the closer the leaf nodes are to the root node, the less fault tolerance can the intermediate nodes provide. So we investigated the distance between the root node and the nearest leaf nodes, and concluded that in graphs with nonzero SLA value, this distance is always 1 or 2. Our GNN was successful in recognizing and exploiting this property.

VII. DISCUSSION AND CONCLUSION

For cloud developers to build reliable services, reference architectures are essential (see Section I). According to the literature, several approaches are introduced for detecting faults focusing on a reduced set of functionalities (e.g. networking or security) by applying a certain set of methods (including machine learning and stochastic modelling) at a well-defined abstraction level (e.g. service mesh or system) as we discussed in Section II.

This paper introduces and demonstrates a feasible generic approach to debug cloud reference architecture candidates in a more automated manner and with less effort even in a dynamic and non-deterministic cloud environment combining a wide set of advanced modelling, debugging and machine learning methods. The selected modelling framework, the PRISM language and model checker has been proven to be an appropriate tool to handle relatively large state spaces, and generate the necessary training datasets incrementally for our experiments and scenarios starting from a simple producer-consumer problem, and ending with complex service meshes (see Section III).

The automatic generation of PRISM models for complex use cases may help significantly accelerate the generation of possible paths in the state space by simulations, contrary to running the deployed services in a real cloud environment (see Sections IV-A, V-A, and VI-A).

The automated identification of suspicious states in the generated state space was performed either through manually defined rules (global predicates) or through deep learning methods (autoencoder), as described in Section IV-B. With autoencoder we achieved promising results during the analysis of cascaded load balancers. We introduced autoencoders to learn the distribution of the normal behaviour. Tests were carried out in four incremental steps. In all four steps, the normal behaviour were able to be clearly separated from erroneous behaviour, based on the reconstruction loss. I.e. erroneous behaviour can be reproduced by the autoencoder much worse than normal behaviour. Consequently, it appears that the learned latent vectors of normal and erroneous behavior are significantly different when generated by the bottleneck layer of the autoencoder.

Concerning LSTM (see Sections V-B), the experiments indirectly approved the nature of the simulation, namely, that the next state is generated in a quasi-random way (based on predefined probabilities). Therefore, LSTM network was unable to identify a pattern in it.

By modelling and verifying our targeted scenarios, we achieved a higher level of test coverage, even the SLA prediction became feasible by applying GNN models (see Sections VI-B). According to our experiments, we were able to train the graph neural network with very low error, and it performed similarly on the hold-out test set for complex service mesh architectures. Thus, the GNNs were able to exploit the additional information provided by the structure of the analyzed graph. The results were further strengthened by a binary classification task, whereas GNNs were able to exploit general properties of the graph to achieve a 100% accuracy rate.

Integrating stochastic modelling and verification techniques with deep learning methods allow us to steer the traversal of state space toward suspicious situations that might contribute to potential cloud system failures in real cloud systems. Such simulation paths to suspicious situations can form an input (i.e. feed) a cloud debugger tool to make an attempt to reproduce the failure in the real cloud environment (see Section I-C) with active control (steering) [8].

There is a good possibility that the predictors and classifiers developed for reference architectures will be widely applicable to real cloud systems, with adjustments or by combining simulation and real data and models.

The following URL contains supplementary materials, including the dataset: `https://github.com/BME-SmartLab/cloud-deep-debug`

VIII. FUTURE WORKS

Considering the research achievements presented, autoencoder and GNN-related results may be applied in practice in the most straightforward manner. For production-level models, further research and development are required, including optimization of hyperparameters, latency, and throughput, as well as the introduction of an inference server.

Furthermore, it would be helpful to examine the explicability of the models using SHAP (Shapley Additive exPlanations) value [66] or layer-wise relevance propagation [67], which will assist in understanding the inner mechanisms of the decision making process.

Our research agenda includes further elaboration of cloud design patterns, such as autoscaling with a variety of policies.

Analysis and prediction based on the combination of event simulation data in correlation with service mesh topology is further to be explored for the purpose of providing practical steering hints for the cloud debugger. The factor of partial observability of service meshes (black box vs. white box), as well as more sophisticated node behaviour strategies would introduce additional real-world properties in our modelling efforts.

The experimental integration of the presented results with a cloud debugger tool [8] is in progress that will provide significant help to cloud service and reference architecture developers.

ACKNOWLEDGMENT

The work reported in this paper, carried out as a collaboration between SZTAKI and BME, has been partly supported by the the European Union project RRF-2.3.1-21-2022-00004 within the framework of the Artificial Intelligence National Laboratory. This work was partially funded by the National Research, Development and Innovation Office (NKFIH) under OTKA Grant Agreement No. K 132838. The presented work of R. Lovas was also supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences. On behalf of the 'MILAB - SmartLab' cloud project, we are grateful for the possibility to use ELKH Cloud [1] which helped us achieve the results published in this paper.

REFERENCES

[1] M. Héder, E. Rigó, D. Medgyesi, R. Lovas, S. Tenczer, F. Török, A. Farkas, M. Emődi, J. Kadlecsik, G. Mezō, Á. Pintér, and P. Kacsuk, "The past, present and future of the ELKH cloud," *Információs Társadalom*, vol. 22, no. 2, p. 128, Aug. 2022.

[2] P. Pääkkönen and D. Pakkala, "Reference architecture and classification of technologies, products and services for big data systems," *Big Data Res.*, vol. 2, no. 4, pp. 166–186, Dec. 2015.

[3] A. C. Marosi, M. Emodi, A. Farkas, R. Lovas, R. Beregi, G. Pedone, B. Nemeth, and P. Gaspar, "Toward reference architectures: A cloud-agnostic data analytics platform empowering autonomous systems," *IEEE Access*, vol. 10, pp. 60658–60673, 2022.

[4] *Microsoft Azure Documentation Reference Architectures*. Accessed: Feb. 20, 2022. [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/browse/

[5] *The TOGAF Standard, Version 9.2 Overview*. Accessed: Feb. 12, 2022. [Online]. Available: https://www.opengroup.org/togaf

[6] *What is a Reference Architecture?—Enterprise it Definitions*. Accessed: Feb. 20, 2022. [Online]. Available: https://www.hpe.com/us/en/what-is/reference-architecture.html

[7] A. Ullah, H. Dagdeviren, R. C. Ariyattu, J. DesLauriers, T. Kiss, and J. Bowden, "MiCADO-edge: Towards an application-level orchestrator for the cloud-to-edge computing continuum," *J. Grid Comput.*, vol. 19, no. 4, p. 47, Dec. 2021.

[8] B. Ligetfalvi, M. Emődi, J. Kovács, and R. Lovas, "Fundamentals of a novel debugging mechanism for orchestrated cloud infrastructures with macrosteps and active control," *Electronics*, vol. 10, no. 24, p. 3108, Dec. 2021.

[9] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *Proc. 23rd Int. Conf. Comput. Aided Verification* (Lecture Notes in Computer Science), vol. 6806. Berlin, Germany: Springer, 2011, pp. 585–591.

[10] G. Aceto, A. Botta, W. de Donato, and A. Pescape, "Cloud monitoring: Definitions, issues and future directions," in *Proc. IEEE 1st Int. Conf. Cloud Netw. (CLOUDNET)*, Nov. 2012, pp. 63–67.

[11] S. U. Farooq, S. Quadri, and N. Ahmad, "Metrics, models and measurements in software reliability," in *Proc. IEEE 10th Int. Symp. Appl. Mach. Intell. Informat. (SAMI)*, Jan. 2012, pp. 441–449.

[12] P. Kumari and P. Kaur, "A survey of fault tolerance in cloud computing," *J. King Saud Univ.-Comput. Inf. Sci.*, vol. 33, no. 10, pp. 1159–1176, 2021.

[13] S. Swetha and S. K. Venkatesh, "Fault detection and prediction in cloud computing," *Int. J. Trend Sci. Res. Develop.*, vol. 2, no. 12, p. 24, 2018.

[14] W. Gao and Y. Zhu, "A cloud computing fault detection method based on deep learning," *J. Comput. Commun.*, vol. 5, no. 12, pp. 24–34, 2017.

[15] M. Smara, M. Aliouat, A.-S.-K. Pathan, and Z. Aliouat, "Acceptance test for fault detection in component-based cloud computing and systems," *Future Gener. Comput. Syst.*, vol. 70, pp. 74–93, May 2017.

[16] A. A. Abbasi, S. Shamshirband, M. A. A. Al-Qaness, A. Abbasi, N. T. AL-Jallad, and A. Mosavi, "Resource-aware network topology management framework," *Acta Polytechnica Hungarica*, vol. 17, no. 4, pp. 89–101, 2020.

[17] P. Zhang, S. Shu, and M. Zhou, "An online fault detection model and strategies based on SVM-grid in clouds," *IEEE/CAA J. Autom. Sinica*, vol. 5, no. 2, pp. 445–456, Mar. 2018.

[18] Y. Tamura, Y. Nobukawa, and S. Yamada, "A method of reliability assessment based on neural network and fault data clustering for cloud with big data," in *Proc. 2nd Int. Conf. Inf. Sci. Secur. (ICISS)*, Dec. 2015, pp. 1–4.

[19] T. Wang, W. Zhang, J. Wei, and H. Zhong, "Fault detection for cloud computing systems with correlation analysis," in *Proc. IFIP/IEEE Int. Symp. Integr. Netw. Manage. (IM)*, May 2015, pp. 652–658.

[20] P. Zhang, S. Shu, and M. Zhou, "Adaptive and dynamic adjustment of fault detection cycles in cloud computing," *IEEE Trans. Ind. Informat.*, vol. 17, no. 1, pp. 20–30, Jan. 2021.

[21] X. Zhang and Y. Zhuang, "A fault detection algorithm for cloud computing using QPSO-based weighted one-class support vector machine," in *Proc. Int. Conf. Algorithms Archit. Parallel Process.*, Cham, Switzerland: Springer, 2019, pp. 286–304.

[22] K. Rusek, J. Suárez-Varela, A. Mestres, P. Barlet-Ros, and A. Cabellos-Aparicio, "Unveiling the potential of graph neural networks for network modeling and optimization in SDN," in *Proc. ACM Symp. SDN Res.*, Apr. 2019, pp. 140–151.

[23] P. Almasan, J. Suárez-Varela, K. Rusek, P. Barlet-Ros, and A. Cabellos-Aparicio, "Deep reinforcement learning meets graph neural networks: Exploring a routing optimization use case," *Comput. Commun.*, vol. 196, pp. 184–194, Dec. 2022.

[24] F. Meng, L. Jagadeesan, and M. Thottan, "Model-based reinforcement learning for service mesh fault resiliency in a web application-level," 2021, *arXiv:2110.13621*.

[25] R. R. Karn, R. Das, D. R. Pant, J. Heikkonen, and R. Kanth, "Automated testing and resilience of Microservice's network-link using istio service mesh," in *Proc. 31st Conf. Open Innov. Assoc. (FRUCT)*, Apr. 2022, pp. 79–88.

[26] P. R. B. N. Tomás, "Using machine learning (ML) for anomaly detection over traffic present in service mesh architectures," Ph.D. thesis, Dept. Inform. Eng., Fac. Sci. Technol., Univ. Coimbra, Coimbra, Portugal, 2022. [Online]. Available: https://estudogeral.uc.pt/bitstream/10316/102166/4/PedroRafaelBarataNinharelhosTom%c3%a1s.pdf

[27] L. Wu, J. Bogatinovski, S. Nedelkoski, J. Tordsson, and O. Kao, "Performance diagnosis in cloud microservices using deep learning," in *Proc. Int. Conf. Service-Oriented Comput.*, H. Hacid, F. Outay, H.-Y. Paik, A. Alloum, M. Petrocchi, M. R. Bouadjenek, A. Beheshti, X. Liu, and A. Maaradji, Eds. Cham, Switzerland: Springer, 2021, pp. 85–96.

[28] F. Asadova, G. Kertesz, R. Lovas, and S. Szenasi, "Fault detection in GPU-enabled cloud systems—An overview," in *Proc. IEEE 20th Jubilee World Symp. Appl. Mach. Intell. Informat. (SAMI)*, Mar. 2022, pp. 317–322.

[29] P. Kacsuk, "Systematic macrostep debugging of message passing parallel programs," *Future Gener. Comput. Syst.*, vol. 16, no. 6, pp. 609–624, Apr. 2000.

[30] R. N. Calheiros, R. Ranjan, C. A. De Rose, and R. Buyya, "CloudSim: A novel framework for modeling and simulation of cloud computing infrastructures and services," 2009, *arXiv:0903.2525*.

[31] N. Mansouri, R. Ghafari, and B. M. H. Zade, "Cloud computing simulators: A comprehensive review," *Simul. Model. Pract. Theory*, vol. 104, Nov. 2020, Art. no. 102144.

[32] D. H. King and S. H. Harrison, "'JaamSim' open-source simulation software," in *Proc. Grand Challenges Modeling Simulation Conf., (GCMS)*, Vista, CA, USA: Society for Modeling & Simulation International, Jul. 2013, pp. 1–6.

[33] N. Matloff, "Introduction to discrete-event simulation and the simpy language," Dept. Comput. Sci., Davis, CA, USA, Jan. 2008, vol. 2. [Online]. Available: https://web.cs.ucdavis.edu/~matloff/matloff/public_html/156/PLN/DESimIntro.pdf

[34] G. F. Riley and T. R. Henderson, "The Ns-3 network simulator," in *Modeling and Tools for Network Simulation*, K. Wehrle, M. Güneş, and J. Gross, Eds. Berlin, Germany: Springer, 2010, pp. 15–34.

[35] *IMUNES IP Network Emulator/Simulator*. Accessed: Sep. 28, 2022. [Online]. Available: http://imunes.net/

[36] A. Bergmayr, U. Breitenbücher, N. Ferry, A. Rossini, A. Solberg, M. Wimmer, G. Kappel, and F. Leymann, "A systematic review of cloud modeling languages," *ACM Comput. Surv.*, vol. 51, no. 1, pp. 1–38, Feb. 2018.

[37] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Boston, MA, USA: Addison-Wesley, 2002.

[38] L. Lamport, "The PlusCal algorithm language," in *Proc. Int. Colloq. Theor. Aspects Comput.* Cham, Switzerland: Springer, 2009, pp. 36–60.

[39] A. Bianco and L. D. Alfaro, "Model checking of probabilistic and nondeterministic systems," in *Proc. Int. Conf. Found. Softw. Technol. Theor. Comput. Sci.* Cham, Switzerland: Springer, 1995, pp. 499–513.

[40] J. Kovacs, G. Kusper, R. Lovas, and W. Schreiner, "Integrating temporal assertions into a parallel debugger," in *Euro-Par 2002 Parallel Processing*, B. Monien and R. Feldmann, Eds. Berlin, Germany: Springer, 2002, pp. 113–120.

[41] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han, "Service mesh: Challenges, state of the art, and future research opportunities," in *Proc. IEEE Int. Conf. Service-Oriented Syst. Eng. (SOSE)*, Apr. 2019, pp. 122–1225.

[42] L. Richardson and S. Ruby, *RESTful Web Services*. Sebastopol, CA, USA: O'Reilly Media, 2008.

[43] M. Xu Chen, O. Firat, A. Bapna, M. Johnson, W. Macherey, G. Foster, L. Jones, N. Parmar, M. Schuster, Z. Chen, Y. Wu, and M. Hughes, "The best of both worlds: Combining recent advances in neural machine translation," 2018, *arXiv:1804.09849*.

[44] X. Lu, Y. Tsao, S. Matsuda, and C. Hori, "Speech enhancement based on deep denoising autoencoder," in *Proc. Interspeech*, Aug. 2013, pp. 436–440.

[45] W. Xu, K. Shawn, and G. Wang, "Adversarially approximated autoencoder for image generation and manipulation," *IEEE Trans. Multimedia*, vol. 21, no. 9, pp. 2387–2396, Sep. 2019.

[46] X. Bampoula, G. Siaterlis, N. Nikolakis, and K. Alexopoulos, "A deep learning model for predictive maintenance in cyber-physical production systems using LSTM autoencoders," *Sensors*, vol. 21, no. 3, p. 972, Feb. 2021.

[47] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, *arXiv:1412.6980*.

[48] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[49] C. Lea, M. D. Flynn, R. Vidal, A. Reiter, and G. D. Hager, "Temporal convolutional networks for action segmentation and detection," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 156–165.

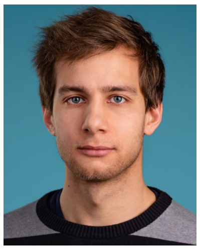
[50] B. Tang and D. S. Matteson, "Probabilistic transformer for time series analysis," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 34, 2021, pp. 23592–23608.

[51] J. Koutník, K. Greff, F. Gomez, and J. Schmidhuber, "A clockwork RNN," 2014, *arXiv:1402.3511*.

[52] D. Neil, M. Pfeiffer, and S.-C. Liu, "Phased LSTM: Accelerating recurrent network training for long or event-based sequences," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 3882–3890.
[53] J. Chung, S. Ahn, and Y. Bengio, "Hierarchical multiscale recurrent neural networks," 2016, *arXiv:1609.01704*.
[54] S. Mehri, K. Kumar, I. Gulrajani, R. Kumar, S. Jain, J. Sotelo, A. Courville, and Y. Bengio, "SampleRNN: An unconditional End-to-End neural audio generation model," 2016, *arXiv:1612.07837*.
[55] S. Merity, N. S. Keskar, and R. Socher, "Regularizing and optimizing LSTM language models," 2017, *arXiv:1708.02182*.
[56] S. Merity, "Single headed attention RNN: Stop thinking with your head," 2019, *arXiv:1911.11423*.
[57] H. Zen, "Acoustic modeling in statistical parametric speech synthesis—From HMM to LSTM-RNN," in *Proc. MLSLP*, 2015, pp. 1–10.
[58] Y. Baek and H. Y. Kim, "ModAugNet: A new forecasting framework for stock market index value with an overfitting prevention LSTM module and a prediction LSTM module," *Expert Syst. Appl.*, vol. 113, pp. 457–480, Dec. 2018.
[59] Z. Karevan and J. A. K. Suykens, "Transductive LSTM for time-series prediction: An application to weather forecasting," *Neural Netw.*, vol. 125, pp. 1–9, May 2020.
[60] *GEXF File Format*. Accessed: Sep. 28, 2022. [Online]. Available: https://gexf.net/
[61] F. Monti, F. Frasca, D. Eynard, D. Mannion, and M. M. Bronstein, "Fake news detection on social media using geometric deep learning," 2019, *arXiv:1902.06673*.
[62] A. Derrow-Pinion, J. She, D. Wong, O. Lange, T. Hester, L. Perez, M. Nunkesser, S. Lee, X. Guo, B. Wiltshire, P. W. Battaglia, V. Gupta, A. Li, Z. Xu, A. Sanchez-Gonzalez, Y. Li, and P. Velickovic, "ETA prediction with graph neural networks in Google maps," in *Proc. 30th ACM Int. Conf. Inf. Knowl. Manage.*, Oct. 2021, pp. 3767–3776.
[63] J. Xiong, Z. Xiong, K. Chen, H. Jiang, and M. Zheng, "Graph neural networks for automated de novo drug design," *Drug Discovery Today*, vol. 26, no. 6, pp. 1382–1393, Jun. 2021.
[64] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch geometric," 2019, *arXiv:1903.02428*.
[65] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" 2018, *arXiv:1810.00826*.
[66] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Proc. Adv. Neural Inf. Process. Syst.*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Red Hook, NY, USA: Curran Associates, 2017, pp. 4765–4774.
[67] W. Samek, G. Montavon, A. Vedaldi, L. K. Hansen, and K.-R. Müller, *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*, vol. 11700. Springer, 2019.

RÓBERT LOVAS received the Ph.D. degree in informatics from the Budapest University of Technology and Economics (BME). He is the Deputy Director at the Institute for Computer Science and Control (SZTAKI), Eotvos Lorand Research Network (ELKH). He is a Habilitated Associate Professor and the Founder of the Institute for Cyber-Physical Systems at the John von Neumann Faculty of Informatics, Obuda University, and a member of the Committee on Information Science at the Hungarian Academy of Sciences. His research and development experience in wide range of application fields of distributed and parallel, systems has been gained in various global, EU and national collaborations with academic organizations, universities, and enterprises focusing on computational chemistry, numerical meteorological modelling, bioinformatics, agriculture, connected cars, and Industry 4.0. He has been coordinating EU FP7/H2020 projects, and the ELKH Cloud research infrastructure. His latest cloud, big data, IoT and AI-related research achievements contribute to the recently launched Artificial Intelligence and Autonomous Systems National Laboratories.

ERNÖ RIGÓ received his MSc degree in systems engineering from the Budapest University of Technology and Economics (BME). He is the Head of the Department of Network Security and Internet Technologies at the Institute for Computer Science and Control (SZTAKI), Eotvos Lorand Research Network (ELKH). He is an Assistant Lecturer at the Institute for Cyber-Physical Systems at the John von Neumann Faculty of Informatics, Obuda University, where he is also a Ph.D. student. As an ISACA Certified Information Systems Auditor (CISA) and ISC2 Certified Information Systems Security Professional (CISSP), his Ph.D. research is focused on security evaluation of dynamic cloud infrastructures. He is responsible for system design, implementation and day to day operation of the ELKH Cloud research infrastructure. His cloud and security related research achievements contribute to the recently launched Artificial Intelligence and Autonomous Systems National Laboratories.



DÁNIEL UNYI received the M.Sc. degree in biomedical engineering (Hons.) in 2021. His master's thesis was concerned with graph variational autoencoders and their applications in bioinformatics. With the ideas and applications introduced in his thesis, he placed 2nd on the National Scientific Student's Competition of Hungary. His Ph.D. research is focused on graph-based representation learning, self-supervised learning, explainable AI, and their practical application in 3D medical image processing. He is a PhD student in Computer Engineering at BME, under the supervision of Balint Gyires-Toth. He also works in multiple industrial projects, which involve the introduction of deep learning through real-world problems rooted in computer vision and network analysis.

BÁLINT GYIRES-TÓTH received the Ph.D. degree (summa cum laude) in January 2014. He is an Associate Professor at BME. He has been conducts research on fundamental and applied machine learning since 2007. His leadership, the first Hungarian hidden Markovmodel based Text-To-Speech (TTS) system was introduced in 2008. Since then, his primary research field is deep learning. His main research interests are sequential data modelling with deep learning, self-supervised learning and deep reinforcement learning. He also participates in applied deep learning projects, including time series modelling, anomaly detection, computer vision and conversational AI. He was involved in various successful research and commercial projects. In 2017, he was certified as NVidia Deep Learning Institute (DLI) Instructor and University Ambassador. His latest AI-related research achievements contribute to the recently launched Artificial Intelligence Systems National Laboratory as a subproject leader.