2024 IEEE International Conference on Service-Oriented System Engineering (SOSE)

Diffusing High-level SLO in Microservice Pipelines

Boris Sedlak, Víctor Casamayor Pujol, Praveen Kumar Donta, and Schahram Dustdar

Distributed Systems Group, Vienna University of Technology (*TU Wien*), Vienna 1040, Austria.
{b.sedlak, v.casamayor, pdonta, dustdar}@dsg.tuwien.ac.at

***Abstract*—Complex interactions within microservice architectures obfuscate the implications of individual services to high-level requirements. This becomes even more grave for multi-tenant and multi-vendor scenarios, like Edge computing, where different stakeholders might specify opposing Service Level Objectives (SLOs), e.g., minimizing both energy consumption and response time. To avoid contradictions within SLOs and to infer how SLOs can be fulfilled, this paper presents a methodology that diffuses high-level SLOs into multiple lower levels of SLOs and parameter assignments. Thus, it becomes clear how individual sub-processes contribute to high-level SLOs, and how these must be configured to foster their fulfillment. We evaluated our methodology for several microservice pipelines, where the challenge is to ensure multiple high-level SLOs (e.g., customer satisfaction) by finding and constraining all influential factors. The results show that by inferring multiple layers of lower-level constraints, we can fulfill high-level SLOs up to 100%. Notably, we could extract that the restrictiveness of low-level SLOs and the occurrence of conflicts have a severe impact on SLO fulfillment.**

***Index Terms*—Service Level Objectives, Microservices, Intelligent Systems, Edge Computing, Requirements Assurance**

I. Introduction

Many current internet-based applications are composed of a network of microservices, each providing a specific functionality to the application; common instances are data transformation pipelines or machine learning pipelines. These instances benefit particularly from service-oriented architecture [1], which improves both modularity and flexibility, while keeping services loosely coupled – boosting scalability. However, each service's performance depends on its neighboring services, i.e., those that send or receive data from it. Hence, if its performance deteriorates, this affects neighboring services and, ultimately, the overall application performance.

To assess the application's overall performance, Cloud computing uses Service Level Objectives (SLOs); typical SLOs are response time or availability, which refers to the entire application, but not to individual services [2]. Whenever an SLO is violated, services are scaled to reestablish the expected performance level. However, Cloud providers are generally unaware of which services should actually be scaled; simply scaling all services (or candidates) can turn out extremely inefficient. In this sense, there exist works that pinpoint which services to scale by finding applications' critical path [3] or performing causal analysis on service architectures [4]. Applying such methods requires considerable time, which can propagate failures in large and distributed applications [5].

However, when looking into novel computing paradigms, such as Edge computing [6] or the computing continuum [7], some Cloud-based rules simply do not apply: First, only parts of their infrastructure can be scaled; secondly, both paradigms assume a multi-tenant and multi-vendor scenario [8], i.e., infrastructure is used to host multiple applications, which belong to different stakeholders. When stakeholders set their SLOs, it is challenging to identify whether these are compatible; in many cases, SLOs of different stakeholders can be opposing – causing conflicts. For instance, infrastructure providers could aim at hosting several applications on devices, and hence limit processing time available per tenant (i.e., the applications); application developers, on the other hand, want maximum quality for end users. Attempts to satisfy both will result in a contradiction, which must be circumvented to avoid undesired system behavior, or even worse, breakdown.

SLOs can be used to constrain different levels of abstraction, from high-level goals such as response time and client satisfaction, down to hardware utilization of individual devices. For application stakeholders, the most intuitive choice is to start posing SLOs that look at the overall performance of the system [9]; we call the resulting constraints "high-level SLOs". These high-level SLOs can target different aspects of QoS or Quality of Experience (QoE), such as high video stream resolution, or decreased energy consumption, but also cost. The question remaining is how to determine under which conditions a system can actually fulfill them. For example, what does it take to minimize energy consumption? The answer might be to restrict CPU load or other resource utilization; we call these derivative constraints "low-level SLOs". However, it is tedious for application developers to specify SLOs for increasingly large microservice applications; in most cases, they would also lack in-depth knowledge of how to diffuse a high-level SLO into the corresponding low-level SLOs.

To decrease the overall complexity of system design, we present a 3-step methodology that diffuses high-level SLOs throughout an application, which means splitting them up into a set of lower-level SLOs. To control all of these SLOs and maintain them within bounds, the methodology identifies parameters that causally influence the required SLO fulfillment, and how they should be assigned. Finally, the methodology detects conflicts caused by high-level SLOs, which might occur at any abstraction level. If possible, these conflicts are resolved autonomously; otherwise, it is indicated to stakeholders that they require amendment. Thus, the contributions of this article are the following:

1) A service-oriented methodology that describes application requirements through multiple layers of SLOs. This enables fine-grained control of the overall system in multi-tenant and multi-vendor scenarios.

2642-6587/24/$31.00 ©2024 IEEE
DOI 10.1109/SOSE62363.2024.00008

2) A diffusion mechanism that propagates high-level SLOs into lower-level ones. To fulfill the associated high-level SLOs, the algorithm defines adequate performance ranges for lower-level SLOs. Further, it identifies parameters (if they exist) that are able to control lower-level SLOs.
3) A conflict identification algorithm for high-level SLOs based on diffused lower-level SLOs. The algorithm is able to resolve conflicts (if they can be solved autonomously), and otherwise alert stakeholders.

The remainder of the paper is structured as follows: Section II introduces background knowledge and related work; Section III presents our methodology for diffusing SLOs, which is implemented and evaluated in Section IV. Finally, Section V, concludes our paper with a future scope.

II. Preliminaries

This section provides an overview of background knowledge on Bayesian networks and how these can be used to specify SLOs. Furthermore, it contains related work that applies SLOs and Bayesian networks for describing system requirements.

A. Background

Bayesian Networks (BNs), as applied by Pearl [10], are structural causal models that can be represented as Directed Acyclic Graph (DAG); see Figure 1 for an example graph that is trained in Section IV-C. Nodes in a BN represent random variables (e.g., cpu), whereas an edge between two variables (e.g., $cpu \rightarrow energy$) indicates a conditional dependency, i.e., cpu has an impact on the states that $energy$ takes. For example, given that we observe a variable assignment $cpu = x$, the BN can provide the probability that a dependent variable is in a respective state $energy = y$; this can be expressed using Bayes' theorem as shown in Eq. (1).

$$P(energy \mid cpu) = \frac{P(cpu \mid energy) \times P(energy)}{P(cpu)} \quad (1)$$

BNs are used to model real-world processes: numerous works (e.g., [11]–[13]) train BNs from historical observations (i.e., metrics) to model the probabilities of different system states. Thus, BNs can answer how likely it is to observe a certain variable assignment, e.g., a system runtime state, given historical observations. Spinning this thought further, in previous work [14], [15] we applied BNs to predict SLO fulfillment of microservices. In particular, we constructed high-level SLOs around BN variables, e.g., $energy \leqslant 10W$, to infer the probability of SLO violations under different assignments of dependent variables. Thus, we found satisfying service configurations (i.e., parameter assignments) by repeatedly evaluating queries like $P(energy > 10 \mid cpu = x)$.

The respective SLO descriptions contain the following three parts: a random variable (var), a conditional relation (rel), and a threshold ($thresh$), where $var \in$ BN, $rel \in \{\leqslant, \geqslant\}$, and $thresh \in \mathbb{Q}$; hence, possible examples are $latency \leqslant 10$, or $cpu \leqslant 95$. The second SLO type requires an objective (obj) and a variable (var), where $obj \in \{min, max\}$; such objectives, e.g., $max(QoE)$, are optimized during operation.

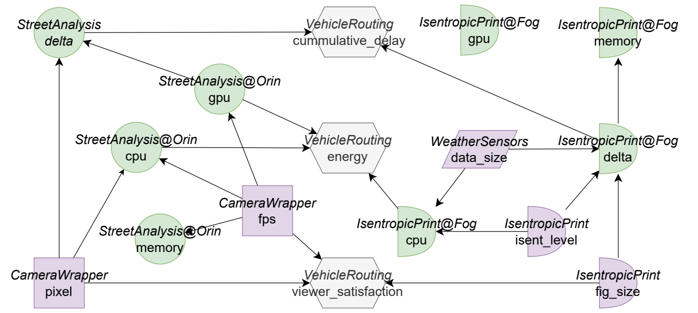

Fig. 1: Combined BN for a microservice pipeline that consists of the following evaluate services: *VehicleRouting* (yellow center), *CameraWrapper* and *StreetAnalysis* (left), and *WeatherSensors* and *IsentropicPrint* (right)

Noteworthy, within [14] we presented `BNL` – a customized algorithm for Bayesian Network Learning (BNL) – which will be applied later in the methodology of this paper.

Given a BN, such as Figure 1, we note two fundamental properties that will be exploited in this paper: (1) any variable (v) that describes a high-level SLO is a leaf node, i.e., it has only incoming edges, otherwise, v's child (or grandchild) would be constrained; since BNs are acyclic, there is always a leaf. Hence, edges in BNs point toward the high-level SLO, which means that fulfilling them is a consequence of maintaining all parent variables in a desired range – these are low-level SLOs. Further, (2) parameters are root nodes, i.e., without incoming edges, because they are conditionally independent of other variables; if there were some, actively setting a parameter would remove any parent edge. The diffusion algorithm in Section III-C will build upon these properties.

B. Related Work

In the context of this paper, we identified two main areas of related work that intersect with our research: (1) SLO-aware service description to continuously ensure system requirements, and (2) modeling systems as large-scale BNs to estimate how changes propagate or can be countered.

1) SLO-Aware Service Description: Pusztai et al. [9], [16], [17] provide *next-level SLO* descriptions, i.e., such that are composed of multiple variable thresholds; hence, SLOs can reflect more complex conditions. Their central contribution – an edge-based workload scheduler – is similar to Guan and Boukerche [18]; the latter present QoS-aware processing methods through different AI methods, though BNs were not discussed. To ensure latency SLOs, Seo et al. [19] provide a dynamic decomposition of ML tasks into smaller subtasks; however, SLOs were not diffused further. Cao [20] outlines a research agenda for an SLO-oriented management layer for cloud-edge infrastructure; Cardelli et al. [21] design an autonomous elasticity mechanism to ensure QoS in cloud-edge service chains. The authors in [22] discuss the importance of controlling distributed systems with *DeepSLOs*, i.e., such that span multiple abstraction layers; their vision, however, was not implemented yet, as counts for [23].

Given these works, we summarize that SLOs are the state-of-the-art solution to specify requirements for cloud comput-

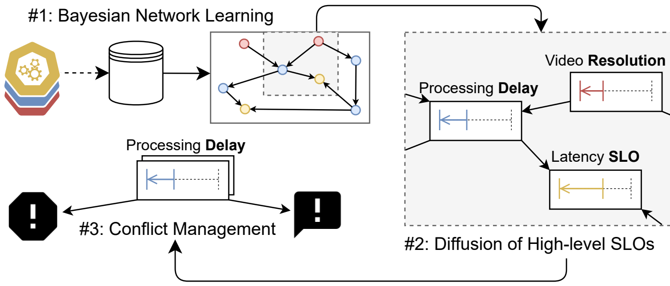

Fig. 2: 3-Step methodology for ensuring high-level SLOs through diffusion

ing; nevertheless, there is an ongoing translation of SLOs from the cloud to the edge. Although authors like [18] and [19] recognize the importance of AI to ensure edge-based SLOs, none of the presented would further diffuse high-level SLOs to identify respective lower-level SLOs.

2) Large-scale Bayesian Network Modelling: Yazdi et al. [11] presented a dynamic BN to assess the resilience of a pipeline system – providing insights under which conditions QoS can be assured. Extending to compound systems, Chen et al. [4] provided a dynamic causality graph called *CauseInfer* that pinpoints issues during runtime. *CauseInfer* uses a two-tier mechanism to split a system into device and service layers. Wang et al. [24] transformed a dynamic fault tree into a BN to trace fault propagation within a vehicle control network. This could infer the probability of faults under different hypothetical setups. BNL is still an actively developing field: Kitson et al. [25] provide a comprehensive overview of techniques and algorithms that create accurate causal models, whereas Vowels et al. [26] provides a survey on that topic.

Given these works, we conclude that numerous works focus on training accurate BNs from observations; the use cases behind them are manifold. Although most of them apply the BN to extract some sort of knowledge, none of them used its conditional dependencies to infer how target states can be assured through lower-level requirements.

III. Methodology

In this section, we start by presenting a set of research questions. Then, we illustrate our 3-step methodology that ensures high-level requirements by disseminating them into lower-level subcomponents; it first trains a BN for a service composition or pipeline, then diffuses low-level SLOs and parameter assignments, and lastly indicates and resolves conflicts within low-level SLOs and parameters. Figure 2 provides an overview of this methodology; the sequential steps are embedded into the respective subsections III-B to III-D.

For all following algorithms, Table I presents a summary of variable notations used in this methodology section.

A. Research Questions

In the following, we describe three research questions extracted from the introduction, each accompanied by a motivating description. These questions will guide both the methodology as well as its evaluation

TABLE I: Frequently used variable notations

Notation	Meaning
s	An individual microservice
M_s	Multidimensional metrics describing s state
M	Wrapper for all metrics in the application
D	Training data set joint for all services
G	Bayesian network graph trained from D
Q	List of all high-level SLOs
q	An individual high-level SLO $q \in Q$
p	The parent node of another variable (e.g., q)
p_p	A grandparent node, i.e., parent of p
S_{hl}	List of desired states to fulfill SLOs
hl	A state of a high-level SLO variable
ll	A state of a lower-level parent variable
ll_q	Total probability of fulfilling q with $p = ll$
X	Dictionary to store ll_q according to ll
t	Probability threshold for including a state ll
λ	Hyperparameter to customize acceptance range
L	List of raw low-level SLOs and parameters
v	A random variable in G, might be q, p, etc
L_v	Duplicate constraints for v in L
k	Intersection between multiple constraints
A	List of constraints without duplicates (easy)
B	List of constraints that presented minor conflicts
U	List of low-level SLOs and assignments (final)
C	List of major conflicts that were not resolved

RQ-1*) How can high-level SLOs be translated to lower-level objectives?* Fulfillment of high-level SLOs emerges from a wider equilibrium among the system components; this can be ensured by maintaining sub-processes (or components) under the respective conditions that foster this. However, to the best of our knowledge, there exist no mechanisms that translate stakeholders' high-level SLOs into lower-level SLOs. As an answer to that, our methodology should infer low-level SLOs by leveraging in-depth knowledge about system dynamics.

RQ-2*) How restrictive should low-level SLOs be?* The more hierarchical and dense a list of SLOs becomes, the less trivial it is what values a low-level SLO should assume to fulfill high-level ones. In reality, predicting the behavior of complex systems will not yield a single possible outcome, but a probabilistic list of states. To that extent, low-level SLOs can hardly be expressed in "black-or-white logic", but the question is how to decide if a low-level state is desirable or not.

RQ-3*) Where do conflicts among SLOs occur and how can they potentially be resolved?* When reasoning rationally, it is intuitive that a system cannot fulfill two competing requirements at the same time, e.g., minimizing *energy* while maximizing *customer_satisfaction*. However, with an increasing number of SLOs, stakeholders cannot always maintain an overview; hence, the question is in which part of the system conflicts will actually occur, and to what extent, or under which conditions, they can be resolved autonomously.

B. Bayesian Network Learning

Given a microservices application, e.g., a sequential processing pipeline, the objective of this first methodology step is to provide a causal understanding of the dependencies between the services. To achieve this, we reveal the relations of different services through BNL – this combines all their variables in one graph. Before that, however, we must collect the necessary

training data. Therefore, we observe all applied microservices during runtime and collect multidimensional metrics (M_s) that describe each service's (s) internal processes.

Training data can be collected periodically or in one operation; in any case, the data from all microservices is combined within D. Notice, that metrics from different services must be captured under equal configurations, e.g., if a pipeline contains two sequential microservices *CameraWrapper* → *StreetAnalysis*, the streaming data produced by *CameraWrapper* must be the exact same received and processed by *StreetAnalysis*. This can be assured by (1) capturing both services' metrics at the same time and joining rows over their timestamp, or (2) maintaining comparable conditions for data sets and joining them over interface variables, i.e., such that describe the same transmitted data on both sides, like *video resolution*.

Next, metrics are processed with BNL to turn them into a composite graph; this is reflected by Algo. 1 (Lines 1-4), which also provides the wrapper for the overall methodology. While `BNL` is known from Section II-A, `JOIN` combines training data of different services incrementally into one data set (D); feeding D to `BNL` turns it into a graph G. Afterward, G contains the dependencies between service variables and their conditional probabilities of assuming certain variable states.

Algorithm 1 Wrapper for the 3 methodology steps

Require: M, Q {Service metrics and high-level SLOs}
Ensure: U, C {Low-level SLOs, params, and conflicts}
1: $L, D \leftarrow \varnothing$
2: **for each** M_s **in** M **do**
3: $\quad D \leftarrow \texttt{JOIN}(D, M_s)$
4: **end for**
5: $G \leftarrow \texttt{BNL}(D)$ { – Step 1}
6: $L \leftarrow \texttt{HLD}(G, Q, \varnothing, \varnothing)$ { – Step 2}
7: $U, C \leftarrow \texttt{CIR}(L)$ { – Step 3}
8: **return** U, C

C. Diffusion of High-level SLOs

The diffusion requires the BN (G) from the previous step and a list of high-level SLOs (Q) – the shape of individual SLOs is as introduced in Section II-A. In the following, we start traversing G from nodes that represent high-level SLOs and then gradually visit their ancestors (i.e., nodes with an edge pointing to them). To fulfill high-level SLOs, each node is extended with a threshold it must ensure; if it is a root node, it is called a "parameter", otherwise a "low-level SLO". This is shown in Fig. 3, where high-level SLO variables are located on the right (purple); by traversing its parents, the middle column is constrained to certain thresholds, which are reached through the parameter assignments on the left (i.e., grandparents). Visiting variables more than one time can lead to conflicts – this is discussed further in Section III-D.

The diffusion's abstract implementation is shown in Algo. 2, which accepts two additional input parameters: a parent node (p) and a list of target states (S_{hl}). However, these are

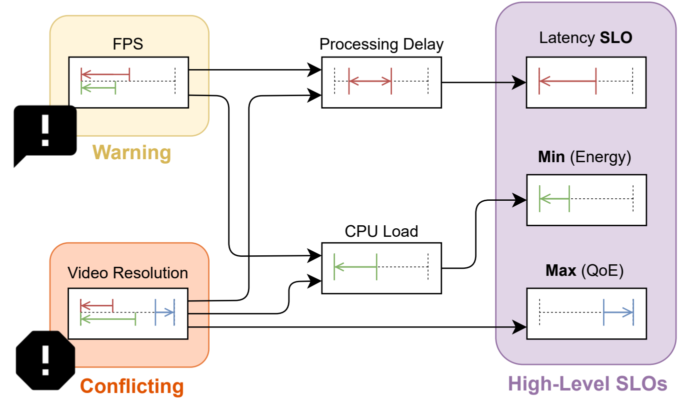

Fig. 3: Diffusing high-level SLOs into lower-level SLOs and assignments

only set in subsequent recursions. The start case (Lines 2-5) is simple: for every high-level SLO (q), find all states $hl \in \texttt{STATES}(G, q)$ that satisfy q's target condition (Line 4). For example, given an SLO $latency \leqslant 10$, S_{hl} summarizes all known states of *latency* that meet this threshold. Next, in Lines 18-20, find q's parents and constrain each parent node (p_p) by inferring respective low-level states that cause S_{hl}.

Traversing q's parents instantiates multiple recursions (Lines 6-14): for every parent variable (p), find p's states that (likely) fulfill q; in other words, this is the low-level SLO or parameter assignment. This can be inferred from a BN through variable elimination [27] (`VE`) – an instance of exact Bayesian inference. For every low-level state $ll \in \texttt{STATES}(G, p)$, we call $\texttt{VE}(G, q, p = ll)$, which returns the probability of different outcomes (i.e., that a high-level state $q = hl$ occurs) when assigning $p = ll$; in Algo. 2, we abbreviate this $P(q = hl \mid p = ll)$ as z. However, if observing $q = hl$ is actually desirable, is determined by hl's occurrence in the list of target states (Line 9). For every state ll, the probability of $p = ll$ causing a desired outcome (i.e., fulfilling q) is summarized (ll_q) and appended to X – a temporary storage to collect these probabilities.

Whether a state ll is included in the low-level SLO, is determined by ll_q – its probability of causing q to be fulfilled. In particular, ll_q must meet the acceptance threshold (t), which is calculated relative to the state with the highest probability (Line 15). The acceptance range can be customized through the hyperparameter $\lambda \in (0, 1]$ – higher λ raises t proportionally; hence, the acceptance range becomes more narrow, meaning fewer states can satisfy it. The accepted states constitute either a low-level SLO or a parameter assignment of p; what follows, is that these constraints are appended to L, as done for recursively visited nodes.

After all high-level SLO variables and their ancestors were visited, L contains all low-level SLOs and parameters that were inferred from G; however, it potentially includes duplicate entries for variables that were visited multiple times. This methodology step addressed **(RQ-1)** by presenting a diffusion mechanism for high-level SLOs; **(RQ-2)** was equally addressed by specifying the acceptance threshold λ. Nevertheless, for both of them, the evaluation must provide further

Algorithm 2 High-level SLO diffusion (HLD)

Require: $G, Q, p, S_{hl}; \lambda$ (global)
Ensure: L {List of low-level SLOs and parameters}
1: **for each** q **in** Q **do**
2: **if** $p = \varnothing \vee S_{hl} = \varnothing$ **then**
3: $p \leftarrow q$
4: $S_{hl} \leftarrow \{hl \mid hl \in \texttt{STATES}(G, q), q(hl) = \texttt{True}\}$
5: **else**
6: **for each** ll **in** $\texttt{STATES}(G, p)$ **do**
7: $ll_q \leftarrow 0$
8: **for each** (hl, z) **in** $\texttt{VE}(G, q, p = ll)$ **do**
9: **if** $hl \in S_{hl}$ **then**
10: $ll_q \leftarrow ll_q + z$
11: **end if**
12: **end for**
13: $X[ll] \leftarrow (ll, ll_q)$
14: **end for**
15: $t \leftarrow max(X) \times \lambda$
16: $S_{hl} \leftarrow \{ll \mid (ll, ll_q) \in X, ll_q \geqslant t\}$
17: **end if**
18: **for each** p_p **in** $\texttt{PARENTS}(G, p)$ **do**
19: $L \leftarrow \texttt{HLD}(G, q, p_p, S_{hl}) \cup L$
20: **end for**
21: **end for**
22: **return** $L \cup (p, S_{hl})$

details on their influence on high-level SLO fulfillment.

D. Conflict Management

After inferring low-level SLOs and parameter assignments, the entire collection (L) is post-processed to identify and resolve conflicts. Generally, if a variable $v \in G$ was visited n times, then L contains n constraints for v – what differs are the imposed thresholds, each according to another high-level SLO. Recall Fig. 3, where the grandparent on the left (i.e., *fps* and *video_res*) were visited two, and respectively, three times; the colored arrows in the variable range indicate from which high-level SLO the constraint originated. The central difference between the two cases is the following: for *fps* there exists a satisfying intersection of its constraints (red $\cap$ green), whereas the constraints of *video_res* are disjoint and not satisfiable. In the following, we resolve the former case as "minor conflict", and indicate the latter as "major conflict".

This behavior is expressed in more detail in Algo. 3; in particular, all entries in L are traversed to determine if there are conflicts, and whether they can be resolved. In the simplest case, a variable (v) is only present once in L; all variables that fulfill this condition are collected in A (Line 4). Otherwise, for a list of duplicate constraints (L_v), the intersection between all the variables' constraints is calculated according to Eq (2).

$$\texttt{INTER}(L_v) = \bigcap_{i,j=1; i \neq j}^{n} L_i \cap L_j \neq \varnothing \tag{2}$$

If there exists an intersection (k), this resolves the conflict and k is appended to B (Line 8) – the list of minor conflicts.

Algorithm 3 Conflict identification and resolution (CIR)

Require: L {List of low-level SLOs and parameters}
Ensure: U, C {Unique constraints and conflicts}
1: $A, B, C \leftarrow \varnothing$
2: **for each** (v, S_{hl}) **in** L **do**
3: **if** $\texttt{COUNT}(L, v) = 1$ **then**
4: $A \leftarrow (v, S_{hl}) \cup A$
5: **else**
6: $k \leftarrow \texttt{INTER}(\texttt{DUPL}(L, v))$
7: **if** $k \neq \varnothing$ **then**
8: $B \leftarrow (v, k) \cup B$
9: **else**
10: $C \leftarrow v \cup C$
11: **end if**
12: **end if**
13: **end for**
14: **return** $A \cup B, C$

Otherwise, if the constraints are disjoint, v is appended to C (Line 10) – the list of major conflicts. Finally, Algo. 3 returns a list of unique constraints (U), which combines $A \cup B$; C is maintained separately so that these conflicts can be indicated to application developers. Algo. 1 returns the same lists.

With the presented methodology, conflicts occur independently of the order in which high-level SLOs are traversed; it is not the case, for example, that the first high-level SLO visiting a variable is prioritized. However, resolving major conflicts would inevitably require some sort of hierarchy among the high-level SLOs, otherwise there cannot be any satisfying variable assignment. Hence, we answered what kinds of conflicts can be resolved **(RQ-3)**; the evaluation will provide further details on where conflicts actually occur. This concluded the presented methodology, which started by training a BN from metrics, inferring low-level SLOs, and finally, in this subsection, resolving conflicts as far as possible, or at least indicating them to the application developer.

IV. Evaluation

To evaluate the ideas presented in this paper, we focus on the individual steps of the methodology and highlight whether the outcome fulfills the research questions. For this, we first outline how the evaluation scenarios were set up and how the methodology was implemented; then we present the results of our experiments and discuss their implications.

A. Evaluation Scenarios

We evaluate our methodology multiple times under different scenarios; each scenario consists of a microservices application, where individual services are chained together to form a composite pipeline. Please refer to Table II for a list of all microservices. The presented services are categorized into three types: (1) *producer* services provide sensor data, (2) *worker* services run data processing, and (3) *consumer* services face clients and determine how the pipeline is perceived; hence, stakeholders would place high-level SLOs for *consumers*.

TABLE II: Microsevices available for evaluation

ID	type	param / var	host
TrafficSensors [28]	Producer	1 / 1	*Xavier*
HistoricDB	Producer	2 / 2	*Server*
CameraWrapper [29]	Producer	2 / 2	*Nano*
WeatherSensors [30]	Producer	1 / 1	*Xavier*
AnomalyDetection [28]	Worker	0 / 5	*Fog*
HistoricProvision	Worker	2 / 7	*Server*
StreetAnalysis [31]	Worker	0 / 4	*Orin*
PrivacyTransform [29]	Worker	0 / 6	*Orin*
IsentropicPrint [30]	Worker	2 / 6	*Fog*
TrafficPrediction	Consumer	0 / 2	*Fog*
VehicleRouting	Consumer	0 / 3	*Orin*
LiveMonitoring	Consumer	0 / 3	*Server*

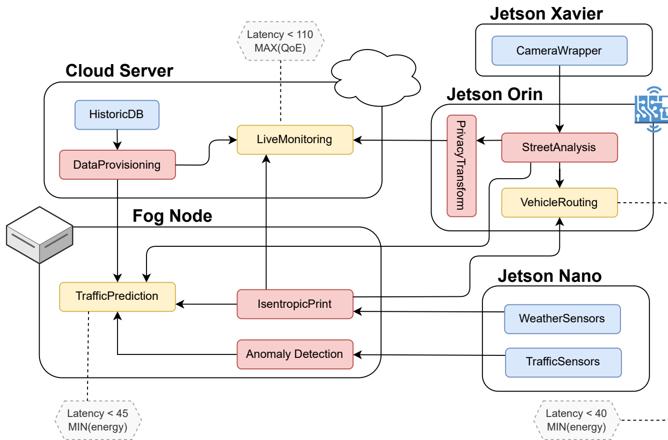

Fig. 4: Logical microservice architecture with the respective hosting devices

The internal state of each service is described by a set of variables, which can be collected and analyzed through metrics. Some variables can actively be set by the stakeholder to change the resulting service; we call those variables "parameters". Each service in Table II features a column that specifies the ratio between parameters and variables. For *IsentropicPrint*, the param/var ratio 2/5 indicates that it features 5 variables, of which 2 are parameters. The last table column specifies at which host[1]the services are executed; please refer to [14] for additional information on device capabilities.

Hosting devices have direct implications on SLO fulfillment, for example, due to heterogeneous hardware capabilities [32]. Among the specified hosts, some devices are more restricted than others; as an example, *Server* dwarfs all Jetson devices (i.e., *Nano*, *Orin*, and *Xavier*). Nevertheless, in this paper, we assume that deployments of individual services are predetermined. Consequently, this also defines the networking delay between services, which in turn, affects the overall execution time for service pipelines distributed over multiple hosts.

In particular, Figure 4 shows the logical distribution between services and hosts, i.e., where individual services are deployed. Microservices are connected alongside the arrows, where data flows in the pointing direction. This creates pipelines from the producers (blue), over the workers (red), to the consumers (yellow). For instance, using *VehicleRouting* and all its parent nodes, it is possible to assemble a smart city application that consumes road conditions to reroute traffic.

In the following, the objective will be to diffuse the high-level SLOs for each consumer application and its dependent services. The grey hexagons in Figure 4 represent high-level SLOs that stakeholders specified for every consumer service. For instance, to evaluate *LiveMonitoring*, its high-level SLOs are diffused over all parent services it depends upon: *IsentropicPrint*, *WeatherSensors*, *PrivacyTransform*, *StreetAnalysis*, and *CameraWrapper*. Thus, we provide evidence of how these services contribute to the posed high-level SLO.

B. Implementation

We provide a Python-based prototype[2]that implements all aspects of our methodology; apart from that, the repository contains all microservices used to generate BNL training data. Notice that, as depicted in Table II, microservices were adopted from existing research as far as possible. As discussed in Section II-A, the applied `BNL` algorithm also originates from previous work; noteworthy, for this paper, we implemented the algorithm with pgmpy [33] – a python framework. To ensure that the trained BN models all applied microservices precisely, the services were configured to evaluate all possible parameter permutations during runtime. While this presents a limitation to the applicability of our methodology, it can be circumvented with alternative approaches, e.g., interpolating between empirically visited configurations [32].

Metrics created by each service are collected in CSV files: 80% are used for BNL, whereas the remaining 20% are retained for evaluation purposes. For each application, we first use the training set to train a BN, which is then used to execute our methodology. Afterward, any resulting SLO fulfillment was measured for the test data set; the scripts to create results, images, or tables are all contained in the repository.

C. Results

In the following, we address the three research questions that were posed in Section III-A. For each question, we explain how it was evaluated, and then discuss the respective results.

1) SLO Diffusion ***(RQ-1)****:* A consequence of successfully translating high-level SLOs to low-level ones would be to find system configurations (i.e., parameter assignments) that fulfill high-level SLOs; hence, we will analyze the resulting SLO fulfillment as an indicator for a correct diffusion. To that extent, we diffuse the respective high-level SLOs over each consumer service and infer low-level SLOs and parameter assignments. We configure the system according to the inferred constraints and analyze whether this could control SLO fulfillment.

The first step is to train a BN for each application; as an example, Figure 1 shows the BN for *VehicleRouting* and all microservices it depends on. Grey nodes reflect high-level SLOs, green ones low-level SLOs, and purple nodes parameters; each service also features a unique symbol. For example,

[2]Prototype artifact available at GitHub, accessed Apr 10th 2024

TABLE III: SLOs and parameters inferred for *VehicleRouting*

Microservice	Variable	States	SLO / Param
VehicleRouting	cumm_delay energy viewer_sat	$\leqslant$ 45 ms $\leqslant$ 19 W ———	High-level
StreetAnalysis *StreetAnalysis* *StreetAnalysis* *IsentropicPrint* *IsentropicPrint*	delta cpu (Orin) gpu (Orin) delta cpu (Fog)	$\leqslant$ 35 ms $\leqslant$ 21 % $\leqslant$ 40 % $\leqslant$ 37 ms $\leqslant$ 17 %	Low-level
CameraWrapper *CameraWrapper* *IsentropicPrint* *IsentropicPrint* *WeatherSensors*	pixel fps fig_size isent_level data_size	= 480 p = 15 f $\leqslant$ 50 p $\leqslant$ 200 k $\leqslant$ 30 pi	Parameter

TABLE IV: High-level SLO fulfillment of inferred and alternative assignment for all three evaluated applications

Microservice	High-level SLO	% Min	% Fulfill	% Max
VehicleRouting	cumm_delay $\leqslant$ 45 min(energy)	0.00 0.53	0.94 0.99	1.00 1.00
TrafficPrediction	cumm_delay $\leqslant$ 40	0.00	0.83	0.90
LiveMonitoring	cumm_delay $\leqslant$ 110 max(viewer_sat.)	0.13 0.00	0.93 1.00	1.00 1.00

the fulfillment of the central *energy* SLO is dependent on the variables that have an edge directed to it, i.e., the *gpu* of *StreetAnalysis*, and the *cpu* from both *StreetAnalysis* and *IsentropicPrint*. Apart from them, there exist nodes that do not impact high-level SLOs (i.e., they have no directed path to grey nodes), which will not be traversed by Algo 2.

The resulting constraints are shown in Table III, which contains all low-level SLOs and parameter values that were diffused from the high-level SLO; notice how *viewer_satisfaction* was not constrained in this scenario. Given the high-level SLOs (i.e., first two rows), the low-level SLOs (i.e., second part) present indicators for preferable variable distributions, which are best assured by assigning parameters as specified. Parameters such as *pixel* and *fps* are assigned to one value, whereas the latter three can assume arbitrary values in a range – each value supposedly fulfills low-level SLOs to a degree $> \lambda$. For instance, any $data_size \leqslant 30$ causes *cpu* and *delta* (check dependencies from Figure 1) to stay in bounds, while keeping *energy* at 19W – the lowest possible assignment.

For all three applications, we configured the parameters according to the inferred thresholds and evaluated the SLO fulfillment; the respective results are contained in Table IV. The maximum (or minimum) values reflect possible values from alternative parameter combinations (i.e., permutations); orange cells indicate cases where our methodology could not infer assignments that maximize high-level SLO fulfillment. These discrepancies occur either due to (1) flexible boundaries, e.g., *fig_size* = 50 is an acceptable assignment $> \lambda$, although *cumm_delay* would be more likely fulfilled with *fig_size* = 10, or (2) conflicts within high-level SLOs, e.g., *LiveMonitoring* cannot ensure both maximum *viewer_satisfaction* and

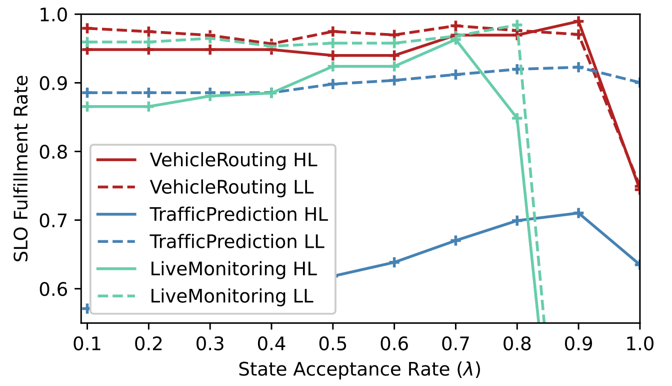

Fig. 5: SLO fulfillment (high/low-level) of different applications and λs

cumm_delay $\leqslant$ 110 for 100 % of observations. Nevertheless, we showed that our approach can reach SLO fulfillment of up to 100 %; the lower bound here was given by the *cumm_delay* SLO of *TrafficPrediction*, which reached 83 %.

*2) Acceptance Range **(RQ-2)**:* The acceptance range (λ) determines the degree of freedom for low-level SLOs and parameter assignments. A narrow margin promises less tolerance for SLO violations but at the same time risks SLO conflicts due to disjoint inference results. To answer what λ is optimal for each application, we vary λ and highlight its effect on high-level and low-level SLO fulfillment.

We apply our methodology with $\lambda \in \{0.1, 0.2, ..., 1.0\}$ and collect the resulting parameter assignments and low-level SLOs. Then, we configure the system according to these constraints and evaluate *all* SLOs: Figure 5 visualizes both the high-level and the low-level SLO fulfillment (dashed or solid lines) for different λ values. The SLO fulfillment (y-axis) is calculated as the average of all microservices included per application, e.g., *VehicleRouting* and all its parents.

We observe, that low-level SLOs are always fulfilled to a higher degree than high-level SLOs, which supports the claim that high-level fulfillment is a consequence. Further, increasing λ had a positive effect on the SLO fulfillment (transition from 0.1 to 0.7); however, as the acceptance range becomes too narrow, *LiveMonitoring* runs into SLO conflicts, indicated by fulfillment = 0. The optimal λ was different for each application; hence, it needs a dynamic mechanism that maximizes λ without risking conflicts.

*3) Conflicts (and Resolution) **(RQ-3)**:* The missing piece for this RQ is to answer where in the BN conflicts actually occur. To that extent, we prepare different combinations of high-level SLOs, provide them to the diffusion algorithm, and analyze for each application whether conflicts occur, and where they occur. Still, whenever possible, conflicts should be resolved automatically, otherwise indicated to stakeholders.

The DAG for *LiveMonitor* equals in large parts the one of *VehicleRouting* in Figure 1, which is why we reuse it for the following explanations. To show how and when conflicts occur, we focus our evaluation on *LiveMonitor*: we provide three high-level SLOs, various thresholds for each of them,

TABLE V: Conflicts among high-level SLOs for *LiveMonitor*

cumm_delay	min(*energy*)	max(*customer_sat*)	both
$\leqslant$ 120 ms	✓	✓	$\lightning$\{*fps*\}
$\leqslant$ 100 ms	$\lightning$\{*pixel*\}	✓	$\lightning\{\wedge \cup$ *pixel*\}
$\leqslant$ 50 ms	$\lightning\{\wedge \cup$ *fps*\}	$\lightning$\{*gpu, pixel, fps*\}	$\lightning\{\wedge \cup$ *gpu*\}
$\leqslant$ 40 ms	$\lightning\{\wedge \cup$ *batch*\}	$\lightning\{\wedge\}$	$\lightning\{\wedge \cup$ *fig_size*\}
$\leqslant$ 25 ms	$\lightning\{\wedge\}$	$\lightning\{\wedge \cup$ *cache_db*\}	$\lightning\{\wedge \cup$ *cache_db*\}

and combine them as depicted in Table V. The *cumm_delay* is always included in the diffusion but combined either with min(*energy*), max(*customer_sat*), or both of them.

Depending on the combinations of high-level SLOs and the threshold of *cumm_delay* $\leqslant \{120, 100, 50, 40, 25\}$, different variables start to show conflicts. In particular, Table V also shows for each combination of high-level SLOs whether it creates any major conflict, which is indicated by a $\lightning$ symbol. While *cumm_delay* $\leqslant$ 120 did not produce conflicts with either min(*energy*) or max(*customer_sat*), applying both immediately causes a conflict for *fps*. In the rows below, smaller *cumm_delay* gradually leads to more conflicts; $\wedge$ indicates that all conflicts from the above line are propagated, hence, $\leqslant 25$ and min(*energy*) led to three conflicting variables. Thus, conflicts can be identified prior to runtime, which is useful to indicate what type of high-level SLOs can be combined.

D. Limitations

When diffusing high-level SLOs, the complexity of Algo. 2 is dominated by the number of `STATES` of high-level SLO variables (h) and their ancestors (l), leading to a complexity of $\mathcal{O}((h \times l \times Q)^m)$. Hence, this approach works well for variables that have few discrete states, or continuous variables that are binned into a low number of bins; how much precision this sacrifices depends on the use case. Apart from that, the complexity is determined by the depth of ancestors (m).

Various optimizations could be applied to Algo. 2, one of them would be to "fold up" longer subtrees in the BN that are single-parented, i.e., do not have other parent nodes. This means, that none of them would have to be extended with a low-level SLO, except for the root node and its direct children (not grandchildren); thus, the list of SLOs could be simplified. However, this condition did not occur in the evaluation, which shows that we must also aim for more complex use cases to improve our methodology further.

Lastly, the algorithm puts a lot of emphasis on the quality of the BN: if G does not accurately reflect reality, e.g., edges are missing or pointing in the wrong direction, the outcome of the algorithm will deviate. Although BN quality was not the focus of this paper, the evaluation indicated that BNs are a bottleneck for the methodology. In particular, the applied techniques often fluctuate regarding the direction of edges. These minor issues can prove fatal for the results of the algorithm, hence, we tried to pin respective edges according to expert knowledge to create a stable evaluation environment. Nevertheless, the results of different BNL techniques and their impact on high-level SLO fulfillment must be the focus of future work.

V. Conclusion & Future Work

This paper presented a diffusion mechanism that translates stakeholders' high-level SLOs into lower-level constraints. For a composition of microservices, it becomes thus clear how individual sub-processes contribute to high-level objectives, and how these must be configured to ensure SLO fulfillment. In particular, we presented a 3-step methodology that infers this knowledge from a Bayesian network, while resolving potential conflicts among competing SLOs as far as possible. The evaluation showed how multiple high-level SLOs, each targeting different QoS or QoE aspects, can be diffused over four different microservice pipelines. For each application, the inferred constraints could exert direct control over high-level SLO fulfillment, which was consequently satisfied between 83 % to 100 % of observations. Further, we could show the impact that the "restrictiveness" of low-level SLO assignments has on higher-level SLOs and how conflicts that occur can endanger these values. In that regard, future work will use these insights to improve the methodology further.

Acknowledgment

Research received funding from the EU's Horizon Europe Research and Innovation Program under Grant Agreement No. 101070186. EU website for Teadal: https://www.teadal.eu/

References

[1] M. Huhns and M. Singh, "Service-oriented computing: key concepts and principles," *IEEE Internet Computing*, vol. 9, no. 1, pp. 75–81, Jan. 2005, conference Name: IEEE Internet Computing.

[2] Z. Zhang, Y. Zhao, and J. Liu, "Octopus: SLO-Aware Progressive Inference Serving via Deep Reinforcement Learning in Multi-tenant Edge Cluster," in *Service-Oriented Computing*, Cham, 2023.

[3] Z. Zhang, M. K. Ramanathan, P. Raj, A. Parwal, T. Sherwood, and M. Chabbi, "{CRISP}: Critical Path Analysis of {Large-Scale} Microservice Architectures," 2022, pp. 655–672.

[4] P. Chen, Y. Qi, and D. Hou, "CauseInfer: Automated End-to-End Performance Diagnosis with Hierarchical Causality Graph in Cloud Environment," *IEEE Transactions on Services Computing*, 2019.

[5] J. Soldani, G. Montesano, and A. Brogi, "What Went Wrong? Explaining Cascading Failures in Microservice-Based Applications," in *Service-Oriented Computing*, J. Barzen, Ed. Cham: Springer International Publishing, 2021, pp. 133–153.

[6] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge Computing: Vision and Challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, Oct. 2016.

[7] S. Dustdar, V. Casamayor Pujol, and P. K. Donta, "On Distributed Computing Continuum Systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 4, pp. 4092–4105, Apr. 2023.

[8] V. Casamayor Pujol, P. K. Donta, A. Morichetta, I. Murturi, and S. Dustdar, "Edge Intelligence—Research Opportunities for Distributed Computing Continuum Systems," *IEEE Internet Computing*, vol. 27, no. 4, pp. 53–74, Jul. 2023, conference Name: IEEE Internet Computing.

[9] S. Nastic, A. Morichetta, T. Pusztai, S. Dustdar, X. Ding, D. Vij, and Y. Xiong, "SLOC: Service Level Objectives for Next Generation Cloud Computing," *IEEE Internet Computing*, vol. 24, no. 3, May 2020.

[10] J. Pearl, "Causal inference in statistics: An overview," *Statistics Surveys*, vol. 3, no. none, pp. 96–146, Jan. 2009.

[11] M. Yazdi, F. Khan, R. Abbassi, and N. Quddus, "Resilience assessment of a subsea pipeline using dynamic Bayesian network," *Journal of Pipeline Science and Engineering*, vol. 2, no. 2, p. 100053, Jun. 2022.

[12] M. Odiathevar, W. K. Seah, and M. Frean, "A Bayesian Approach To Distributed Anomaly Detection In Edge AI Networks," *IEEE Transactions on Parallel and Distributed Systems*, Dec. 2022.

[13] M. Toğaçar, "Detecting attacks on IoT devices with probabilistic Bayesian neural networks and hunger games search optimization approaches," *Transactions on Telecommunications Technologies*, 2022.

[14] B. Sedlak, V. C. Pujol, P. K. Donta, and S. Dustdar, "Designing Reconfigurable Intelligent Systems with Markov Blankets," in *Service-Oriented Computing*, 2023, pp. 42–50.

[15] B. Sedlak, V. Casamayor Pujol, P. K. Donta, and S. Dustdar, "Controlling Data Gravity and Data Friction: From Metrics to Multidimensional Elasticity Strategies," in *IEEE SSE 2023*, Chicago, IL, USA, Jul. 2023.

[16] T. Pusztai, A. Morichetta, V. C. Pujol, S. Dustdar, S. Nastic, X. Ding, D. Vij, and Y. Xiong, "SLO Script: A Novel Language for Implementing Complex Cloud-Native Elasticity-Driven SLOs," in *2021 IEEE ICWS*. Chicago, IL, USA: IEEE, Sep. 2021, pp. 21–31.

[17] T. Pusztai, S. Nastic, A. Morichetta, V. C. Pujol, P. Raith, S. Dustdar, D. Vij, Y. Xiong, and Z. Zhang, "Polaris Scheduler: SLO- and Topology-aware Microservices Scheduling at the Edge," in *15th International Conference on Utility and Cloud Computing*, Dec. 2022, pp. 61–70.

[18] S. Guan and A. Boukerche, "Intelligent Edge-Based Service Provisioning Using Smart Cloudlets, Fog and Mobile Edges," *IEEE Network*, vol. 36, no. 2, pp. 139–145, Mar. 2022.

[19] W. Seo, S. Cha, Y. Kim, J. Huh, and J. Park, "SLO-Aware Inference Scheduler for Heterogeneous Processors in Edge Platforms," *ACM Transactions on Architecture and Code Optimization*, vol. 18, no. 4, pp. 1–26, Dec. 2021.

[20] Y. Cao, "Better Orchestration for SLO-Oriented Cross-site Microservices in Multi-tenant Cloud/Edge Continuum," in *Proceedings of the 24th International Middleware Conference*, New York, USA, Dec. 2023.

[21] V. Cardellini, T. Galinac Grbac, M. Nardelli, N. Tanković, and H.-L. Truong, "QoS-Based Elasticity for Service Chains in Distributed Edge Cloud Environments," in *Autonomous Control*, 2018.

[22] V. C. Pujol and S. Dustdar, "Towards a Prime Directive of SLOs," in *2023 IEEE International Conference on Software Services Engineering (SSE)*, Jul. 2023, pp. 61–70.

[23] J. Walter, D. Okanović, and S. Kounev, "Mapping of Service Level Objectives to Performance Queries," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ser. ICPE '17 Companion. New York, NY, USA: Association for Computing Machinery, Apr. 2017, pp. 197–202.

[24] C. Wang, L. Wang, H. Chen, Y. Yang, and Y. Li, "Fault Diagnosis of Train Network Control Management System Based on Dynamic Fault Tree and Bayesian Network," *IEEE Access*, vol. 9, pp. 2618–2632, 2021.

[25] N. K. Kitson, A. C. Constantinou, Z. Guo, Y. Liu, and K. Chobtham, "A survey of Bayesian Network structure learning," *Artificial Intelligence Review*, vol. 56, no. 8, pp. 8721–8814, Aug. 2023.

[26] M. J. Vowels, N. C. Camgoz, and R. Bowden, "D'ya like DAGs? A Survey on Structure Learning and Causal Discovery," Mar. 2021.

[27] N. Zhang and D. Poole, "A simple approach to Bayesian network computations," in *Engineering-Economic Systems, Stanford*, 1994.

[28] I. G. Wambui, "Improving Traffic Flow Using LSTM Networks in Python: A Step-by-Step Guide," Aug. 2023.

[29] B. Sedlak, I. Murturi, P. K. Donta, and S. Dustdar, "A Privacy Enforcing Framework for Transforming Data Streams on the Edge," *IEEE Transactions on Emerging Topics in Computing*, 2023.

[30] R. May, S. Arms, P. Marsh, E. Bruning, J. Leeman, K. Goebbert, J. Thielen, Z. Bruick, and M. D. Camron, "MetPy: A Python Package for Meteorological Data," Apr. 2024.

[31] B. Sedlak, V. Casamayor Pujol, P. K. Donta, and S. Dustdar, "Markov Blanket Composition of SLOs," in *2024 IEEE International Conference on Edge Computing and Communications (EDGE)*, Shenzhen, China, Jul. 2024.

[32] B. Sedlak, V. C. Pujol, P. K. Donta, and S. Dustdar, "Equilibrium in the Computing Continuum through Active Inference," *Future Generation Computer Systems*, May 2024.

[33] A. Ankan and J. Textor, "pgmpy: A Python Toolkit for Bayesian Networks," Apr. 2023.